# Spring Tutorial - Spring

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/index.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0035__Spring_3_JavaConfig.htm)

## Maven + Spring hello world

In command prompt, issue following Maven command :

mvn archetype:generate -DgroupId=com.java2s.common -DartifactId=Java2sExamples -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Maven will generate all the Java's standard folders structure.

To convert the newly generated Maven style project to Eclipse's style project.

mvn eclipse:eclipse

Then we can import the converted project into Eclipse IDE.

## Spring dependency

Add Spring dependency in Maven's pom.xml file.

<!-- Spring framework -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

<version>2.5.6</version>

</dependency>

Full 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/maven-v4\_0\_0.xsd"**>

<modelVersion>4.0.0</modelVersion>

<groupId>com.java2s.common</groupId>

<artifactId>Java2sExamples</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>Java2sExamples</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- Spring framework -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

<version>2.5.6</version>

</dependency>

</dependencies>

</project>

Issue "mvn eclipse:eclipse" again to download the Spring dependency libraries automatically and put it into your Maven's local repository.

Maven will add the downloaded libraries into Eclipse ".classpath" for dependency purpose.

## Spring bean

Spring's bean is just a normal Java class. It has fields with getter and setter.

Later we will declare in Spring bean configuration file.

Add the following code to "src/main/java/com/java2s/common/HelloWorld.java".

package com.java2s.common;

**public** **class** HelloWorld {

**private** String name;

**public** void setName(String name) {

this.name = name;

}

**public** void printHello() {

System.out.println(**"Spring 3 : Hello ! "** + name);

}

}

## Spring bean configuration file

Next we will create Spring configuration file. In the Spring's bean configuration file, we declare all the available Spring beans.

Create an xml file with name as Spring-Module.xml at "src/main/resources/Spring-Module.xml".

<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-3.0.xsd">

<bean id=**"helloBean"** **class**=**"com.java2s.common.HelloWorld"**>

<property name=**"name"** value=**"java2s"** />

</bean>

</beans>

The previous file shows a typical Spring ApplicationContext configuration. First, Spring's namespaces are declared, and the default namespace is beans.

The beans namespace is used to declare the beans that need to be managed by Spring, and its dependency requirements for Spring to resolve and inject those dependencies.

Afterward, we declare the bean with the ID provider and the corresponding implementation class.

When Spring sees this bean definition during ApplicationContext initialization, it will instantiate the class and store it with the specified ID.

Change the App.java as follows. In the App.java, it loads the Spring bean configuration file (Spring-Module.xml) and retrieve the Spring bean via getBean() method.

It casts the results from the getBean method then call its printHello() method.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

HelloWorld obj = (HelloWorld) context.getBean(**"helloBean"**);

obj.printHello();

}

}

Compile the source code

mvn compile

Run the code above with the following command.

mvn exec:java -Dexec.mainClass=**"com.java2s.common.App"**

The code above generates the following result.

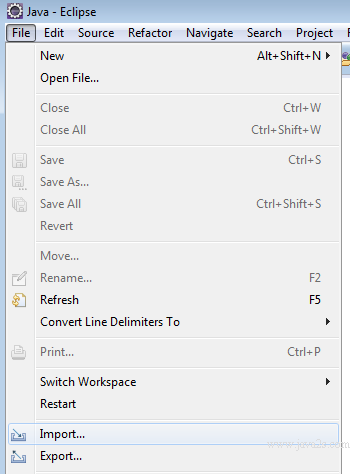
http://www.java2s.com/Tutorials/JavaImage/myResult/S/SPRING_BEAN_CONFIGURATION_FILE__8C530F027E5BC8FC3DC6.PNG   
  
[Download Java2s\_Spring\_XML.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_XML.zip)

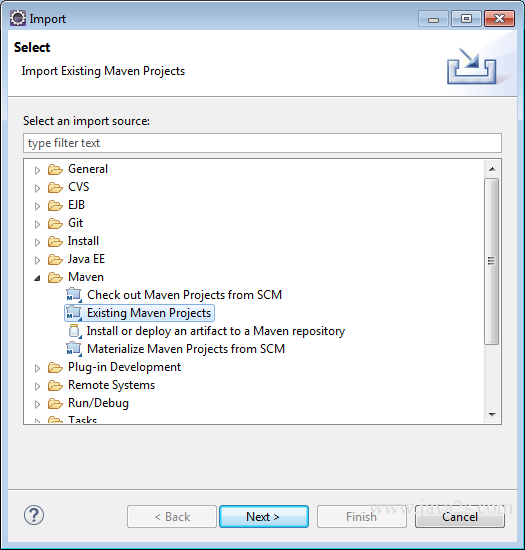
## Setup Eclipse

The following screenshots show how to setup eclipse ide to work with Spring.

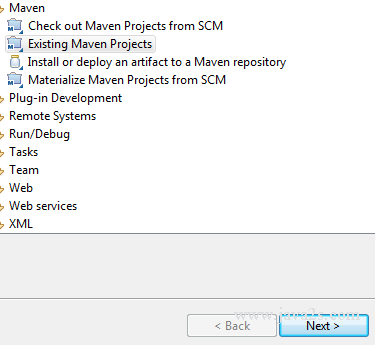
Import the pom.xml to Eclipse



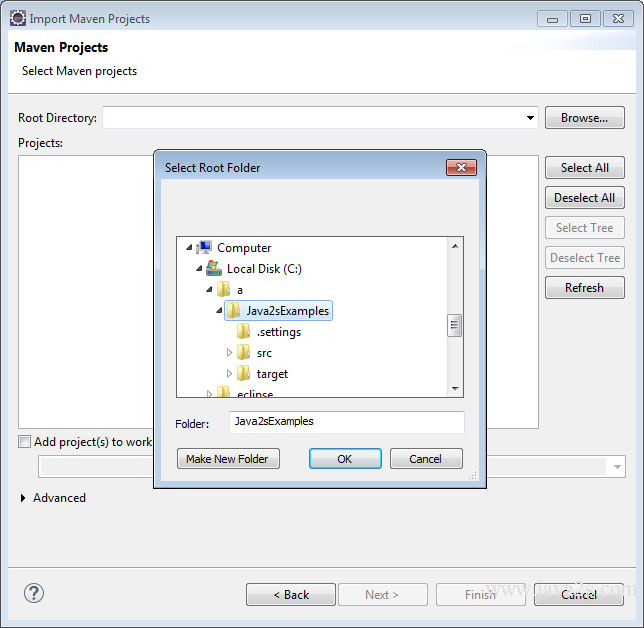
Choose existing maven project.



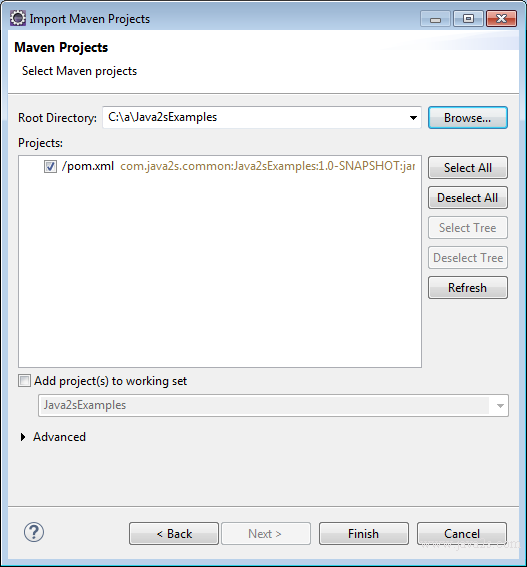
Click Next



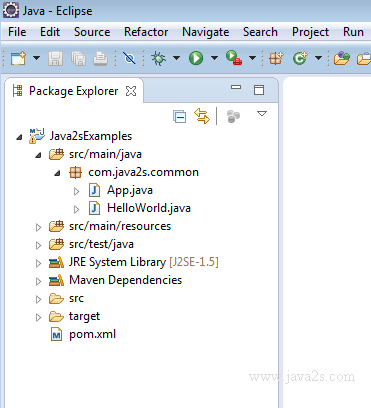
Click browse



Select project



Finish



## Spring 3 hello world

Use below Maven command to create a standard Java project structure.

mvn archetype:generate -DgroupId=com.java2s.common -DartifactId=Java2sExamples -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Use the following command to convert Maven style project to Eclipse's style project, and import the project into Eclipse IDE.

mvn eclipse:eclipse

## Spring 3.0 dependency

Locate the pom.xml and add the Spring 3.0 dependencies listed below.

The Spring dependencies are available for download via Maven central repository.

<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/maven-v4\_0\_0.xsd"**>

<modelVersion>4.0.0</modelVersion>

<groupId>com.java2s.common</groupId>

<artifactId>Java2sExamples</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>Java2sExamples</name>

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

<properties>

<spring.version>3.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- Spring 3 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>

</project>

## Spring bean for Spring 3

The following code contains a simple Spring bean.

package com.java2s.common;

**public** **class** HelloWorld {

**private** String name;

**public** void setName(String name) {

this.name = name;

}

**public** void printHello() {

System.out.println(**"Spring 3 : Hello ! "** + name);

}

}

##### Spring bean configuration file

The following xml file is for SpringBeans.xml. It is a Spring configuration file, and declares all the available Spring beans.

<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-3.0.xsd">

<bean id=**"helloBean"** **class**=**"com.java2s.common.HelloWorld"**>

<property name=**"name"** value=**"java2s"** />

</bean>

</beans>

Modify the App.java as follows and run code.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

HelloWorld obj = (HelloWorld) context.getBean(**"helloBean"**);

obj.printHello();

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/S/SPRING_BEAN_FOR_SPRING_3__F11BFCAA7C99053DEB86.PNG

# Spring Tutorial - Spring JavaConfig

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0030__Spring_HelloWorld.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0040__Spring_Loosely_Coupled.htm)

The following sections show how to use the annotation to mark dependencies for Spring

## Spring JavaConfig Dependency

To use JavaConfig (@Configuration), you need to include CGLIB library.

<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/maven-v4\_0\_0.xsd"**>

<modelVersion>4.0.0</modelVersion>

<groupId>com.java2s.common</groupId>

<artifactId>Java2sExamples</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>Java2sExamples</name>

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

<properties>

<spring.version>3.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- Spring 3 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>

<!-- JavaConfig need this library -->

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>2.2.2</version>

</dependency>

</dependencies>

</project>

## Java Bean

Create a a simple bean as follows.

package com.java2s.common;

**public** **interface** HelloWorld {

void printHelloWorld(String msg);

}

Provide implementation for the interface.

package com.java2s.common;

**public** **class** HelloWorldImpl **implements** HelloWorld {

**public** void printHelloWorld(String msg) {

System.out.println(**"Hello : "** + msg);

}

}

The following code shows how to use use @Configuration to tell Spring that this is the core Spring configuration file, and define bean via @Bean.

package com.java2s.common;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** AppConfig {

@Bean(name=**"helloBean"**)

**public** HelloWorld helloWorld() {

return **new** HelloWorldImpl();

}

}

## Main method

In order to load the our JavaConfig class use AnnotationConfigApplicationContext.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.class);

HelloWorld obj = (HelloWorld) context.getBean(**"helloBean"**);

obj.printHelloWorld(**"Spring3 Java Config"**);

}

}

Run the code above with the following command.

mvn exec:java -Dexec.mainClass=**"com.java2s.common.App"**

Output

http://www.java2s.com/Tutorials/JavaImage/myResult/M/MAIN_METHOD__E56C9AADC85F0D4592BE.PNG   
  
[Download Java2s\_Spring\_JavaConfig.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_JavaConfig.zip)

## Spring 3 JavaConfig @Import example

Two simple Spring beans.

File : Customer.java

package com.java2s.common;

**public** **class** Customer {

**public** void printMsg(String msg) {

System.out.println(**"Customer : "** + msg);

}

}

File : Employee.java

package com.java2s.common;

**public** **class** Employee {

**public** void printMsg(String msg) {

System.out.println(**"Employee : "** + msg);

}

}

Now, use JavaConfig @Configuration to declare above beans.

File : CustomerConfig.java

package com.java2s.common;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** CustomerConfig {

@Bean(name=**"customer"**)

**public** Customer customer(){

return **new** Customer();

}

}

File : EmployeeConfig.java

package com.java2s.common;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** EmployeeConfig {

@Bean(name=**"employee"**)

**public** Employee employee(){

return **new** Employee();

}

}

Use @Import to load multiple configuration files.

File : AppConfig.java

package com.java2s.common;

org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

@Configuration

@Import({ CustomerConfig.class, EmployeeConfig.class })

**public** **class** AppConfig {

}

Load the main configuration file , and test it.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.java2s.config.AppConfig;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** AnnotationConfigApplicationContext(

AppConfig.class);

Customer customer = (Customer) context.getBean(**"customer"**);

customer.printMsg(**"Hello 1"**);

Employee employee = (Employee) context.getBean(**"employee"**);

employee.printMsg(**"Hello 2"**);

}

}

Output



# Spring Tutorial - Spring Loosely Coupled

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0035__Spring_3_JavaConfig.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0050__Spring_Setter_Injection.htm)

In the following we will see how Spring would help us to achieve loosely coupled code.

## Project Code

We would create a project which has several classes. Th project would output data in different format. We can choose to output the data in CSV format or JSON format.

First we will create a Java interface for print.

package com.java2s.output;

**public** **interface** Printer

{

**public** void print();

}

After that we would create CSV printer which will output data in CSV format. The CSV printer implements the Printer interface.

package com.java2s.output.impl;

import com.java2s.output.Printer;

**public** **class** CSVPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Csv Output Printer"**);

}

}

Then it is time to create JSON printer which will output message in JSON format. The JSON printer also implements the Printer interface.

package com.java2s.output.impl;

import com.java2s.output.Printer;

**public** **class** JSONPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Json Output Printer"**);

}

}

## Couple Code

We have several ways to use the CSVPrinter or JSONPrinter. First we can call it directly.

package com.java2s.common;

import com.java2s.output.Printer;

import com.java2s.output.impl.CSVPrinter;

**public** **class** App

{

**public** **static** void main( String[] args )

{

Printer output = **new** CSVPrinter();

output.print();

}

}

It is easy to create CSVPrinter in this way. But we will have to change the source code if we want to switch to JSONPrinter we will have to change the source code and recompile.

For the code above it is easy to change since it has two lines of code. Suppose we have thousands of thousands of code and the CSVPrinter had been declared hundreds of time.

## Spring Way

By using Spring Dependency Injection (DI) we can declare Java Beans in the Spring configuration xml file. Then wire the Java Beans in the xml file. In this way Spring can make our printer loosely coupled to the different Printer implementations.

We change the Helper class to accept the Printer.

package com.java2s.output;

import com.java2s.output.Printer;

**public** **class** OutputHelper

{

Printer outputGenerator;

**public** void print(){

outputGenerator.print();

}

**public** void setOutputGenerator(Printer outputGenerator){

this.outputGenerator = outputGenerator;

}

}

Then we have to create a Spring bean configuration file and declare all your Java object dependencies here.

<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-2.5.xsd">

<bean id=**"OutputHelper"** **class**=**"com.java2s.output.OutputHelper"**>

<property name=**"outputGenerator"** ref=**"csvPrinter"** />

</bean>

<bean id=**"csvPrinter"** **class**=**"com.java2s.output.impl.CSVPrinter"** />

<bean id=**"jsonPrinter"** **class**=**"com.java2s.output.impl.JSONPrinter"** />

</beans>

Call it via Spring

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.java2s.output.OutputHelper;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"Spring-Common.xml"**});

OutputHelper output = (OutputHelper)context.getBean(**"OutputHelper"**);

output.print();

}

}

To switch Printer, we just need to change the Spring XML file for a different Printer. When Printer changed, we need to modify the Spring XML file only.

# Spring Tutorial - Spring Setter Injection

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0040__Spring_Loosely_Coupled.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0060__Spring_Constructor_Injection.htm)

Spring Setter injection is the most popular and simple dependency injection method, it will injects the dependency via a setter method.

In Setter Dependency Injection , the IoC container injects a Java Bean's dependencies via JavaBean-style setter methods.

A Java Bean's setters expose the dependencies the IoC container can manage.

In practice, Setter Injection is the most widely used injection mechanism, and it is one of the simplest IoC mechanisms to implement.

## Example

Suppose we have the following interface and Java beans defined.

package com.java2s.common;

**public** **interface** Printer

{

**public** void print();

}

After that we would create CSV printer which will output data in CSV format. The CSV printer implements the Printer interface.

package com.java2s.common;

**public** **class** CSVPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Csv Output Printer"**);

}

}

Then it is time to create JSON printer which will output message in JSON format. The JSON printer also implements the Printer interface.

package com.java2s.common;

**public** **class** JSONPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Json Output Printer"**);

}

}

By using Spring Dependency Injection (DI) we can declare Java Beans in the Spring configuration xml file. Then wire the Java Beans in the xml file. In this way Spring can make our printer loosely coupled to the different Printer implementations.

We change the Helper class to accept the Printer.

package com.java2s.common;

**public** **class** OutputHelper

{

Printer outputGenerator;

**public** void print(){

outputGenerator.print();

}

**public** void setOutputGenerator(Printer outputGenerator){

this.outputGenerator = outputGenerator;

}

}

## XML Configuration

Then we have to create a Spring bean configuration file and declare all your Java object dependencies here.

<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-2.5.xsd">

<bean id=**"outputHelper"** **class**=**"com.java2s.common.OutputHelper"**>

<property name=**"outputGenerator"** ref=**"csvPrinter"** />

</bean>

<bean id=**"csvPrinter"** **class**=**"com.java2s.common.CSVPrinter"** />

<bean id=**"jsonPrinter"** **class**=**"com.java2s.common.JSONPrinter"** />

</beans>

The following two bean tags declared two Java Beans in Spring configuration xml file.

After the declaration we can use the id value to reference the Java Beans.

<bean id=**"csvPrinter"** **class**=**"com.java2s.common.CSVPrinter"** />

<bean id=**"jsonPrinter"** **class**=**"com.java2s.common.JSONPrinter"** />

The following xml bean tag declared the OutputHelper and inject the dependency via setter injection by using property tag.

<bean id=**"outputHelper"** **class**=**"com.java2s.common.OutputHelper"**>

<property name=**"outputGenerator"** ref=**"csvPrinter"** />

</bean>

In the code above we just injected a 'com.java2s.common.CSVPrinter' bean into 'OutputHelper' object via a setter method setOutputGenerator.

## Load configuration and Run

The following code shows how to use load the configuration and run it.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

OutputHelper output = (OutputHelper)context.getBean(**"outputHelper"**);

output.print();

}

}

Output

http://www.java2s.com/Tutorials/JavaImage/myResult/L/LOAD_CONFIGURATION_AND_RUN__E30DD0CD237BF74B6108.PNG   
  
[Download Java2s\_Spring\_Setter\_Injection.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Setter_Injection.zip)

# Spring Tutorial - Spring Constructor Injection

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0050__Spring_Setter_Injection.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0070__Spring_Ref_Bean.htm)

We can inject dependency via a constructor.

Constructor Dependency Injection occurs when a Java Beans's dependencies are provided to it in its constructor(s).

The Java Bean declares a constructor or a set of constructors, taking its dependencies from parameters, and the IoC container passes the dependencies to the Java Bean when instantiation occurs.

## Example

Suppose we have the following interface and Java beans defined.

package com.java2s.common;

**public** **interface** Printer

{

**public** void print();

}

After that we would create CSV printer which will output data in CSV format. The CSV printer implements the Printer interface.

package com.java2s.common;

**public** **class** CSVPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Csv Output Printer"**);

}

}

Then it is time to create JSON printer which will output message in JSON format. The JSON printer also implements the Printer interface.

package com.java2s.common;

**public** **class** JSONPrinter **implements** Printer

{

**public** void print(){

System.out.println(**"Json Output Printer"**);

}

}

A helper class with a constructor.

package com.java2s.common;

**public** **class** OutputHelper

{

Printer outputGenerator;

**public** OutputHelper(){

}

**public** OutputHelper(Printer p){

this.outputGenerator = p;

}

**public** void print(){

outputGenerator.print();

}

}

The following Spring bean configuration file declares the beans and set the dependency via constructor injection by using constructor-arg tag.

<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-2.5.xsd">

<bean id=**"outputHelper"** **class**=**"com.java2s.common.OutputHelper"**>

<constructor-arg>

<bean **class**=**"com.java2s.common.CSVPrinter"** />

</constructor-arg>

</bean>

<bean id=**"csvPrinter"** **class**=**"com.java2s.common.CSVPrinter"** />

<bean id=**"jsonPrinter"** **class**=**"com.java2s.common.JSONPrinter"** />

</beans>

## Load configuration and Run

The following code shows how to load the Spring configuration file and run the application.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

OutputHelper output = (OutputHelper)context.getBean(**"outputHelper"**);

output.print();

}

}

Output

http://www.java2s.com/Tutorials/JavaImage/myResult/L/LOAD_CONFIGURATION_AND_RUN__E4B7C0AF6FDAE52F42ED.PNG   
  
[Download Java2s\_Spring\_Constructor\_Injection.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Constructor_Injection.zip)

## Constructor injection type ambiguities

The following Employee class has two constructor methods. Both of them accept 3 arguments with different data type.

package com.java2s.common;

**public** **class** Employee

{

**private** String name;

**private** String address;

**private** int age;

**public** Employee(String name, String address, int age) {

this.name = name;

this.address = address;

this.age = age;

}

**public** Employee(String name, int age, String address) {

this.name = name;

this.age = age;

this.address = address;

}

**public** String toString(){

return **" name : "** +name + **" address : "** + address + **" age : "** + age;

}

}

The first one has the following parameters

**public** Employee(String name, String address, int age)

The second one is defined as follows.

**public** Employee(String name, int age, String address)

The age and address switched position.

When creating the Employee object in the Spring configuration xml file we useconstructor-arg tag to provide values for the parameter in the constructor.

In the following Spring bean configuration file, we passed in 'java2s' for name, '1000' for address and '28' for age.

<myPreCode>

<beans ...

<bean id=**"myEmployee"** **class**=**"com.java2s.common.Employee"**>

<constructor-arg>

<value>java2s</value>

</constructor-arg>

<constructor-arg>

<value>1000</value>

</constructor-arg>

<constructor-arg>

<value>28</value>

</constructor-arg>

</bean>

</beans>

Here is the code to run the configuration we set above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"Spring-Employee.xml"**});

Employee cust = (Employee)context.getBean(**"myEmployee"**);

System.out.println(cust);

}

}

Output

http://www.java2s.com/Tutorials/JavaImage/myResult/C/CONSTRUCTOR_INJECTION_TYPE_AMBIGUITIES__2D54E3886936DC0D1D33.PNG

From the result string we can see that the second constructor is called rather than the first constructor.

Spring converts the argument '1000' to int, and then calls take the second constructor, even we passed in the value as a String type.

If Spring cannot find the proper constructor to use, it will prompt following error message

constructor arguments specified but no matching constructor

found in bean **'myEmployee'** (hint: specify index **and**/**or**

type arguments **for** simple parameters to avoid type ambiguities)

In order to match the parameter type in the constructor, we can specify the data type for constructor, via type attribute in the constructor-arg tag.

<beans ...

<bean id=**"myEmployee"** **class**=**"com.java2s.common.Employee"**>

<constructor-arg type=**"java.lang.String"**>

<value>java2s</value>

</constructor-arg>

<constructor-arg type=**"java.lang.String"**>

<value>1000</value>

</constructor-arg>

<constructor-arg type=**"int"**>

<value>28</value>

</constructor-arg>

</bean>

</beans>

Run it again.

http://www.java2s.com/Tutorials/JavaImage/myResult/C/CONSTRUCTOR_INJECTION_TYPE_AMBIGUITIES__38374C8CDEB77026778C.PNG

# Spring Tutorial - Spring Ref Bean

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0060__Spring_Constructor_Injection.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0080__Spring_Bean_Properties.htm)

In a big project we may have several Spring configuration files. One Java bean is defined in one Spring xml configuration file can be referenced in another configuration file viaref tag.

The ref tag has the following syntax.

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

In the following Spring-Output.xml file we created two Java Beans and gave them id.

<beans ...

<bean id=**"CSVPrinter"** **class**=**"com.java2s.output.impl.CSVPrinter"** />

<bean id=**"JSONPrinter"** **class**=**"com.java2s.output.impl.JSONPrinter"** />

</beans>

In the following Spring-Common.xml file we defined a com.java2s.output.PrinterHelper Java Bean and gave it id as PrinterHelper. In order to inject the CSVPrinter defined in the Spring-Output.xml file we have to use ref tag to include it.

<beans ...

<bean id=**"PrinterHelper"** **class**=**"com.java2s.output.PrinterHelper"**>

<property name=**"outputGenerator"** >

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

</property>

</bean>

</beans>

## Reference Local Bean

To reference bean we defined in the same xml file we can use ref tag with localattribute.

It has the following format.

<ref local=**"someBean"**/>

In the following xml code, the bean "PrinterHelper" declared in 'Spring-Common.xml' can access "CSVPrinter" or "JSONPrinter" which are defined in the same file with ref local.

<beans ...

<bean id=**"PrinterHelper"** **class**=**"com.java2s.output.PrinterHelper"**>

<property name=**"outputGenerator"** >

<ref local=**"CSVPrinter"**/>

</property>

</bean>

<bean id=**"CSVPrinter"** **class**=**"com.java2s.output.impl.CSVPrinter"** />

<bean id=**"JSONPrinter"** **class**=**"com.java2s.output.impl.JSONPrinter"** />

</beans>

By using ref local we increase the readability of the xml configuration file.

# Spring Tutorial - Spring Bean Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0070__Spring_Ref_Bean.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0090__Spring_Multiple_Configuration_File.htm)

We can fill data to a Java Bean defined in the Spring configuration xml in several ways.

The following sections show to inject value to the name and type properties defined in the MyClass.

**package** com.java2s.common

**public** **class** MyClass {

**private** String name;

**private** String type;

**public** String getName() {

**return** name;/\* w ww . j av a 2 s . co m\*/

}

**public** **void** setName(String name) {

this.name = name;

}

**public** String getType() {

**return** type;

}

**public** **void** setType(String type) {

this.type = type;

}

}

## property tag and value tag

The following code shows how to inject value within a 'value' tag and enclosed with 'property' tag.

<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-2.5.xsd">

<bean id=**"myClass"** **class**=**"com.java2s.common.MyClass"**>

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

<value>java2s</value>

</property>

<property name=**"type"**>

<value>txt</value>

</property>

</bean>

</beans>

After loading the myClass from Spring configuration xml file the name and type properties are set to java2s and txt respectively.

## Shortcut property tag

We can use shortcut property tag to fill value to Java bean properties in the following way.

The property tag can have a value attribute. We put our value there.

<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-2.5.xsd">

<bean id=**"MyClass"** **class**=**"com.java2s.common.MyClass"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"type"** value=**"txt"** />

</bean>

</beans>

## "p" schema

We can even fill the properties when declaring the Java Bean in the bean tag.

<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-2.5.xsd">

<bean id=**"MyClass"** **class**=**"com.java2s.common.MyClass"**

p:name=**"java2s"** p:type=**"txt"** />

</beans>

In order to use the p schema we have to declare thexmlns:p="http://www.springframework.org/schema/p in the Spring XML bean configuration file.

# Spring Tutorial - Spring Multiple Configuration File

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0080__Spring_Bean_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0100__Spring_Inner_Bean_Inject.htm)

In a large project, we may have more than one Spring's bean configuration files. Put every single bean definition in a single file is hard to maintain. And they may be stored in different folder structures.

For example, we may have a Spring-Common.xml in common folder, a Spring-Connection.xml in connection folder, and a Spring-ModuleA.xml in ModuleA folder.

## Load one by one

One way to load the configuration files is to load them one by one.

For example, we put all above three xml files in classpath

project-classpath/Spring-Common.xml

project-classpath/Spring-Connection.xml

project-classpath/Spring-ModuleA.xml

Then we can use the following code to load multiple Spring bean configuration files.

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"Spring-Common.xml"**,

**"Spring-Connection.xml"**,**"Spring-ModuleA.xml"**});

The code above pass in all file names one by one to theClassPathXmlApplicationContext. The problem is that we have to change the code if we need to add new file name.

## import xml file

Spring allows us to organize all Spring bean configuration files into a single XML file.

In order to host all configuration file we created a new Spring-All-Module.xml file, and import the other Spring bean files.

<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-2.5.xsd">

<import resource=**"common/Spring-Common.xml"**/>

<import resource=**"connection/Spring-Connection.xml"**/>

<import resource=**"moduleA/Spring-ModuleA.xml"**/>

</beans>

Put this file under project classpath.

project-classpath/Spring-All-Module.xml

We can load a single xml file like this :

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**"Spring-All-Module.xml"**);

# Spring Tutorial - Spring Inner Bean Inject

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0090__Spring_Multiple_Configuration_File.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0110__Spring_Bean_Scope.htm)

The following sections show how to use Spring inner bean.

## Inner Java Beans

In the following we have the Customer bean. In the customer object we have a custom object Person.

package com.java2s.common;

**public** **class** Customer

{

**private** Person person;

**public** Customer() {

}

**public** Customer(Person person) {

this.person = person;

}

**public** void setPerson(Person person) {

this.person = person;

}

@Override

**public** String toString() {

return **"Customer [person="** + person + **"]"**;

}

}

Here is the definition for Person class.

package com.java2s.common;

**public** **class** Person

{

**private** String name;

**private** String address;

**private** int age;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

@Override

**public** String toString() {

return **"Person [address="** + address + **",age="** + age + **", name="** + name + **"]"**;

}

}

## Reference Inner Java Beans

One way to inject Person Java bean to Customer bean is to use ref attribute as follows.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"person"** ref=**"PersonBean"** />

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address1"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

[Download Java2s\_Spring\_Ref\_Attr.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Ref_Attr.zip)

The Person bean is defined in the same level with Customer bean and we use ref attribute to reference the Person bean.

If the Person bean is only used in Customer bean we can nest the definition of Person bean inside the Customer bean.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"person"**>

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address1"** />

<property name=**"age"** value=**"28"** />

</bean>

</property>

</bean>

</beans>

[Download Java\_Spring\_Nest\_Bean.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java_Spring_Nest_Bean.zip)

The following xml code shows how to use inner bean for constructor injection.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<constructor-arg>

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address1"** />

<property name=**"age"** value=**"28"** />

</bean>

</constructor-arg>

</bean>

</beans>

[Download Java2s\_Spring\_Inner\_Bean\_Constructor.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Inner_Bean_Constructor.zip)

The following code shows how to run the configuration above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

Output

http://www.java2s.com/Tutorials/JavaImage/myResult/R/REFERENCE_INNER_JAVA_BEANS__C50F5C0E60C0ECB48917.PNG

# Spring Tutorial - Spring Bean Scope

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0100__Spring_Inner_Bean_Inject.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0120__Spring_List_Properties.htm)

When defining Java bean in Spring xml configuration or using the Spring annotation we can mark a Java bean as singleton or prototype.

If a Java bean is marked as singleton, each time we call to get the bean we get the same instance.

If a Java bean is marked as prototype, each time we will get a new instance.

## Java Bean

The following code defines a Java bean with a String type property.

package com.java2s.common;

**public** **class** MyService

{

String message;

**public** String getMessage() {

return message;

}

**public** void setMessage(String message) {

this.message = message;

}

}

## Singleton

If no bean scope is specified in bean configuration file, default to singleton.

<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-2.5.xsd">

<bean id=**"customerService"**

**class**=**"com.java2s.common.MyService"** />

</beans>

Here is the code to run the xml configuration defined above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

MyService customerA = (MyService)context.getBean(**"customerService"**);

customerA.setMessage(**"Message by customerA"**);

System.out.println(**"Message : "** + customerA.getMessage());

//retrieve it again

MyService custB = (MyService)context.getBean(**"customerService"**);

System.out.println(**"Message : "** + custB.getMessage());

}

}

Output

Message : Message by customerA

Message : Message by customerA

The code above get MyService twice from the ApplicationContext. And it sets a String value for the message after the first call.

From the output we can see that both getMessage() call return the same message, which means the ApplicationContext uses one copy of the MyService.

If a bean is a singleton scope in Spring IoC container, getBean() will always return the same instance.

[Download Java2s\_Spring\_Singleton\_Scope.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Singleton_Scope.zip)

## Prototype

To get a new 'customerService' bean instance every time we call getBean(), use prototype scope.

<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-2.5.xsd">

<bean id=**"customerService"** **class**=**"com.java2s.common.MyService"**

scope=**"prototype"**/>

</beans>

Run it again



From the output we can see that the second call to thecontext.getBean("customerService"); we got a new instance of MyService.

[Download Java2s\_Spring\_Prototype\_Scope.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Prototype_Scope.zip)

## Bean scopes annotation

The following code shows how to use the annotation to mark the bean scope.

package com.java2s.common;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;

@Service

@Scope(**"prototype"**)

**public** **class** MyService {

String message;

**public** String getMessage() {

return message;

}

**public** void setMessage(String message) {

this.message = message;

}

}

The following code shows how to enable auto component scanning.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package=**"com.java2s.common"** />

</beans>

Here is the new App.java class to run the configuration above. The customerService is changed to myService. Spring automatically creates a bean name from the class name by lower case the first letter.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

MyService customerA = (MyService)context.getBean(**"myService"**);

customerA.setMessage(**"Message by customerA"**);

System.out.println(**"Message : "** + customerA.getMessage());

//retrieve it again

MyService custB = (MyService)context.getBean(**"myService"**);

System.out.println(**"Message : "** + custB.getMessage());

}

}

Run it again

   
  
[Download Java2s\_Spring\_Beans\_Scope\_Annotation.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Beans_Scope_Annotation.zip)

# Spring Tutorial - Spring List Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0110__Spring_Bean_Scope.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0130__Spring_Set_Properties.htm)

We can fill value or a list of values to a Java bean defined in Spring xml configuration file.

The following sections shows how to fill data to List.

## Java Bean

In order to show how to use xml configuration file to fill collection properties, we defined a Customer object with four collection properties.

package com.java2s.common;

import java.util.ArrayList;

import java.util.List;

**public** **class** Customer

{

**private** List<Object> lists = **new** ArrayList<Object>();

**public** String toString(){

return lists.toString();

}

**public** List<Object> getLists() {

return lists;

}

**public** void setLists(List<Object> lists) {

this.lists = lists;

}

}

Person Java Bean

package com.java2s.common;

**public** **class** Person {

**private** String name;

**private** int age;

**private** String address;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

@Override

**public** String toString() {

return **"Person [name="** + name + **", age="** + age + **", address="** + address

+ **"]"**;

}

}

## List

The following code shows how to fill data to a java.util.List typed property.

The code fills in three values. The first one is hard-coded value 1. The second one is a bean reference. We have to define PersonBean somewhere in order to use it here. The third one is a bean definition with property-setting.

...

<property name=**"lists"**>

<list>

<value>1</value>

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

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sList"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</list>

</property>

...

## Example

Full Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<!-- java.util.List -->

<property name=**"lists"**>

<list>

<value>1</value>

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

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sList"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</list>

</property>

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s1"** />

<property name=**"address"** value=**"address 1"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

Here is the code to load and run the configuration.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

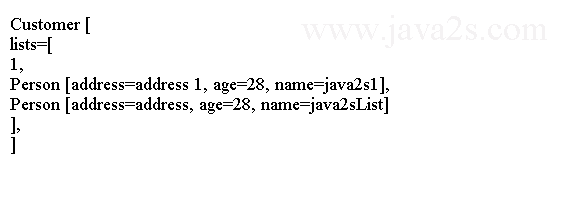
Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

Output

   
  
[Download Java2s\_Spring\_List\_Properties.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_List_Properties.zip)

## ListFactoryBean

ListFactoryBean class can create a concrete List collection class with ArrayList or LinkedList in Spring's bean configuration file. Then we can inject the created list object to our Java bean.

Here is the Java bean class.

package com.java2s.common;

import java.util.ArrayList;

import java.util.List;

**public** **class** Customer

{

**private** List<Object> lists = **new** ArrayList<Object>();

**public** String toString(){

return lists.toString();

}

**public** List<Object> getLists() {

return lists;

}

**public** void setLists(List<Object> lists) {

this.lists = lists;

}

}

Here is Spring's bean configuration file.

<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-2.5.xsd">**

/\*from w w w . ja va 2s . c o m\*/

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"lists"**>

<bean **class**=**"org.springframework.beans.factory.config.ListFactoryBean"**>

<property name=**"targetListClass"**>

<value>java.util.ArrayList</value>

</property>

<property name=**"sourceList"**>

<list>

<value>1</value>

<value>2</value>

<value>3</value>

</list>

</property>

</bean>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_ListFactoryBean.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_ListFactoryBean.zip)

## util schema

We also can use util schema and <util:list> to fill data to a list.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/util/spring-util-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"lists"**>

<util:list list-**class**=**"java.util.ArrayList"**>

<value>1</value>

<value>2</value>

<value>3</value>

</util:list>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_util\_List.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_util_List.zip)

Here is the code to run.

**package** com.java2s.common;

//from w w w . j a v a2s . co m

**import** org.springframework.context.ApplicationContext;

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

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

Customer cust = (Customer) context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

# Spring Tutorial - Spring Set Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0120__Spring_List_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0140__Spring_Map_Properties.htm)

We can fill value or a list of values to a Java bean defined in Spring xml configuration file.

The following sections show how to fill data to Set.

## Java Bean

In order to show how to use xml configuration file to fill collection properties, we defined a Customer object with four collection properties.

package com.java2s.common;

import java.util.HashSet;

import java.util.Set;

**public** **class** Customer {

**private** Set<Object> sets = **new** HashSet<Object>();

**public** Set<Object> getSets() {

return sets;

}

**public** void setSets(Set<Object> sets) {

this.sets = sets;

}

**public** String toString() {

return sets.toString();

}

}

Person Java Bean

package com.java2s.common;

**public** **class** Person {

**private** String name;

**private** int age;

**private** String address;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

@Override

**public** String toString() {

return **"Person [name="** + name + **", age="** + age + **", address="** + address

+ **"]"**;

}

}

## Set

The following code shows how to fill data to a java.util.Set typed property.

The code fills in three values. The first one is hard-coded value 1. The second one is a bean reference. We have to define PersonBean somewhere in order to use it here. The third one is a bean definition with property-setting.

...

<property name=**"sets"**>

<set>

<value>1</value>

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

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sSet"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</set>

</property>

...

## Example

Full Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<!-- java.util.Set -->

<property name=**"sets"**>

<set>

<value>1</value>

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

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sSet"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</set>

</property>

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s1"** />

<property name=**"address"** value=**"address 1"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

Here is the code to load and run the configuration.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

Output

Customer [

sets=[

1,

Person [address=address 1, age=28, name=java2s1],

Person [address=address, age=28, name=java2sSet]]

]

[Download Java2s\_Spring\_Set\_Properties.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Set_Properties.zip)

## SetFactoryBean

SetFactoryBean class can create a concrete Set collection HashSet or TreeSet in Spring's bean configuration file.

The following code shows how to use SetFactoryBean.

Here is the Java bean class.

**package** com.java2s.common;

//from w ww .j a v a2s.c o m

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** Customer {

**private** Set<Object> sets = **new** HashSet<Object>();

**public** Set<Object> getSets() {

**return** sets;

}

**public** **void** setSets(Set<Object> sets) {

this.sets = sets;

}

**public** String toString() {

**return** sets.toString();

}

}

Here is the Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"sets"**>

<bean **class**=**"org.springframework.beans.factory.config.SetFactoryBean"**>

<property name=**"targetSetClass"**>

<value>java.util.HashSet</value>

</property>

<property name=**"sourceSet"**>

<list>

<value>1</value>

<value>2</value>

<value>3</value>

</list>

</property>

</bean>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_SetFactoryBean.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_SetFactoryBean.zip)

## util schema

We also can use util schema and <util:set> to fill data to java.util.Set.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/util/spring-util-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"sets"**>

<util:set set-**class**=**"java.util.HashSet"**>

<value>1</value>

<value>2</value>

<value>3</value>

</util:set>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_set\_util.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_set_util.zip)

Use the following code to run the application.

**package** com.java2s.common;

//from w w w. java 2 s . co m

**import** org.springframework.context.ApplicationContext;

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

**public** **class** App

{

**public** **static** **void** main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/U/UTIL_SCHEMA__24120E5135FB258F8E2F.PNG

# Spring Tutorial - Spring Map Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0130__Spring_Set_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0150__Spring_Bean_Properties_Collection.htm)

We can fill value or a list of values to a Java bean defined in Spring xml configuration file.

The following sections shows how to fill data to Map.

## Java Bean

In order to show how to use xml configuration file to fill collection properties, we defined a Customer object with four collection properties.

package com.java2s.common;

import java.util.HashMap;

import java.util.Map;

**public** **class** Customer {

**private** Map<Object, Object> maps = **new** HashMap<Object, Object>();

**public** String toString() {

return maps.toString();

}

**public** Map<Object, Object> getMaps() {

return maps;

}

**public** void setMaps(Map<Object, Object> maps) {

this.maps = maps;

}

}

Here is the Person bean.

package com.java2s.common;

**public** **class** Person {

**private** String name;

**private** int age;

**private** String address;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

@Override

**public** String toString() {

return **"Person [name="** + name + **", age="** + age + **", address="** + address

+ **"]"**;

}

}

## Map

The following code shows how to fill data to a java.util.Set typed property.

The code fills in three values. The first one is hard-coded value 1. The second one is a bean reference. We have to define PersonBean somewhere in order to use it here. The third one is a bean definition with property-setting. When filling data for java.util.Map we have to provide key and value pair.

...

<property name=**"maps"**>

<map>

<entry key=**"Key 1"** value=**"1"** />

<entry key=**"Key 2"** value-ref=**"PersonBean"** />

<entry key=**"Key 3"**>

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sMap"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</entry>

</map>

</property>

...

## Example

Full Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<!-- java.util.Map -->

<property name=**"maps"**>

<map>

<entry key=**"Key 1"** value=**"1"** />

<entry key=**"Key 2"** value-ref=**"PersonBean"** />

<entry key=**"Key 3"**>

<bean **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2sMap"** />

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

<property name=**"age"** value=**"28"** />

</bean>

</entry>

</map>

</property>

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s1"** />

<property name=**"address"** value=**"address 1"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

Here is the code to load and run the configuration.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

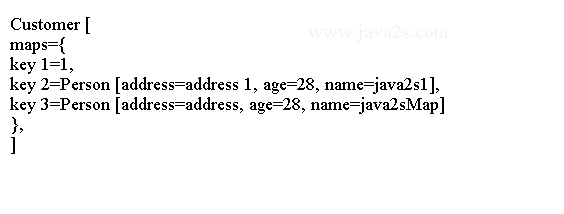
Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

Output

   
  
[Download Java2s\_Spring\_Map\_Properties.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Map_Properties.zip)

## MapFactoryBean

MapFactoryBean class can create a Map collection class HashMap or TreeMap in Spring's bean configuration file.

The following code shows how to create a HashMap, fill data and then inject it into a bean property.

Here is the Java bean class.

**package** com.java2s.common;

//from w w w .j a va2s. c o m

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Customer {

**private** Map<Object, Object> maps = **new** HashMap<Object, Object>();

**public** String toString() {

**return** maps.toString();

}

**public** Map<Object, Object> getMaps() {

**return** maps;

}

**public** **void** setMaps(Map<Object, Object> maps) {

this.maps = maps;

}

}

Here is Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"maps"**>

<bean **class**=**"org.springframework.beans.factory.config.MapFactoryBean"**>

<property name=**"targetMapClass"**>

<value>java.util.HashMap</value>

</property>

<property name=**"sourceMap"**>

<map>

<entry key=**"Key1"** value=**"1"** />

<entry key=**"Key2"** value=**"2"** />

<entry key=**"Key3"** value=**"3"** />

</map>

</property>

</bean>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_MapFactoryBean.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_MapFactoryBean.zip)

## util schema

We also can use util schema and <util:map> to fill data to java.util.Map.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/util/spring-util-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"maps"**>

<util:map map-**class**=**"java.util.HashMap"**>

<entry key=**"Key1"** value=**"1"** />

<entry key=**"Key2"** value=**"2"** />

<entry key=**"Key3"** value=**"3"** />

</util:map>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_Map\_util.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Map_util.zip)

Here is the code to run the application.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/U/UTIL_SCHEMA__0829756474983D271122.PNG

# Spring Tutorial - Spring Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0140__Spring_Map_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0160__Spring_Date_Properties.htm)

We can fill value or a list of values to a Java bean defined in Spring xml configuration file.

The following sections shows how to fill data to java.util.Properties.

## Java Bean

In order to show how to use xml configuration file to fill collection properties, we defined a Customer object with four collection properties.

package com.java2s.common;

import java.util.Properties;

**public** **class** Customer

{

**private** Properties pros = **new** Properties();

**public** Properties getPros() {

return pros;

}

**public** void setPros(Properties pros) {

this.pros = pros;

}

**public** String toString(){

return pros.toString();

}

}

Person Java Bean

package com.java2s.common;

**public** **class** Person {

**private** String name;

**private** int age;

**private** String address;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

@Override

**public** String toString() {

return **"Person [name="** + name + **", age="** + age + **", address="** + address

+ **"]"**;

}

}

## Properties

java.util.Properties is a key-value pair structrue and we can use the following syntax to fill data for it.

...

<property name=**"pros"**>

<props>

<prop key=**"admin"**>user a</prop>

<prop key=**"support"**>user b</prop>

</props>

</property>

...

## Example

Full Spring's bean configuration file.

<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-2.5.xsd">

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<!-- java.util.Properties -->

<property name=**"pros"**>

<props>

<prop key=**"admin"**>user a</prop>

<prop key=**"support"**>user b</prop>

</props>

</property>

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s1"** />

<property name=**"address"** value=**"address 1"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

Here is the code to load and run the configuration.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Customer cust = (Customer)context.getBean(**"CustomerBean"**);

System.out.println(cust);

}

}

Output

Customer [

pros={admin=user a, support=user b},

[Download Java2s\_Spring\_Properties\_Collection.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Properties_Collection.zip)

# Spring Tutorial - Spring Date Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0150__Spring_Bean_Properties_Collection.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0170__Spring_PlaceHolder_Properties.htm)

The following sections shows how to fill data to java.util.Date type value.

## Java Bean

Here is the Java bean class we are going to use.

**package** com.java2s.common;

**import** java.util.Date;

//from w w w . j a va2s . c o m

**public** **class** Customer {

Date date;

**public** Date getDate() {

**return** date;

}

**public** **void** setDate(Date date) {

this.date = date;

}

@Override

**public** String toString() {

**return** **"Customer [date="** + date + **"]"**;

}

}

## Factory bean

The following code shows how to parse a String value to a Date value and then set to Java bean.

It create a Java bean from java.text.SimpleDateFormat and parse in the format asyyyy-MM-dd. Then it use the dateFormat bean as factory-bean and use the parsemethod as the factory-method.

In the constructor-arg tag it sets the value to the date properties.

<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-2.5.xsd">

<bean id=**"dateFormat"** **class**=**"java.text.SimpleDateFormat"**>

<constructor-arg value=**"yyyy-MM-dd"** />

</bean>

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"**>

<property name=**"date"**>

<bean factory-bean=**"dateFormat"** factory-method=**"parse"**>

<constructor-arg value=**"2010-01-31"** />

</bean>

</property>

</bean>

</beans>

[Download Java2s\_Spring\_DateFactory.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_DateFactory.zip)

## CustomDateEditor

In the second method to fill date value, we use the CustomDateEditor class.

In the bean xml configuration file it declares a CustomDateEditor class to convert String into java.util.Date.

<bean id=**"dateEditor"**

**class**=**"org.springframework.beans.propertyeditors.CustomDateEditor"**>

<constructor-arg>

<bean **class**=**"java.text.SimpleDateFormat"**>

<constructor-arg value=**"yyyy-MM-dd"** />

</bean>

</constructor-arg>

<constructor-arg value=**"true"** />

</bean>

Then it declares CustomEditorConfigurer to make Spring convert bean properties whose type is java.util.Date.

<bean **class**=**"org.springframework.beans.factory.config.CustomEditorConfigurer"**>

<property name=**"customEditors"**>

<map>

<entry key=**"java.util.Date"**>

<ref local=**"dateEditor"** />

</entry>

</map>

</property>

</bean>

Here is the full full example of bean configuration file.

<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-2.5.xsd">

<bean id=**"dateEditor"**

**class**=**"org.springframework.beans.propertyeditors.CustomDateEditor"**>

<constructor-arg>

<bean **class**=**"java.text.SimpleDateFormat"**>

<constructor-arg value=**"yyyy-MM-dd"** />

</bean>

</constructor-arg>

<constructor-arg value=**"true"** />

</bean>

<bean **class**=**"org.springframework.beans.factory.config.CustomEditorConfigurer"**>

<property name=**"customEditors"**>

<map>

<entry key=**"java.util.Date"**>

<ref local=**"dateEditor"** />

</entry>

</map>

</property>

</bean>

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"**>

<property name=**"date"** value=**"2010-02-31"** />

</bean>

</beans>

To run the application.

**package** com.java2s.common;

**import** org.springframework.context.ApplicationContext;

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

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

**"SpringBeans.xml"**);

Customer cust = (Customer) context.getBean(**"customer"**);

System.out.println(cust);//from w ww. ja v a2 s .co m

}

}

[Download Java2s\_Date\_CustomDateEditor.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Date_CustomDateEditor.zip)

# Spring Tutorial - Spring Place Holder Properties

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0160__Spring_Date_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0180__Spring_Bean_Inheritance.htm)

PropertyPlaceholderConfigurer class can externalize the deployment details into a properties file.

We can then access its value from bean configuration file via a special format:${variable}.

The following code show show to put database connection properties in a separate file.

## Database properties file

The following code create sa properties file (database.properties). It includes database connection details. We can put it into the project class path.

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/java2sjava

jdbc.username=root

jdbc.password=password

The following xml configuration file use PropertyPlaceholderConfigurer to map the 'database.properties' properties file .

<bean **class**=**"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>

<property name=**"location"**>

<value>database.properties</value>

</property>

</bean>

Here is the full xml configuration file.

<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-2.5.xsd">

<bean **class**=**"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>

<property name=**"location"**>

<value>database.properties</value>

</property>

</bean>

<bean id=**"customerDAO"** **class**=**"com.java2s.customer.dao.impl.JdbcCustomerDAO"**>

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

</bean>

<bean id=**"customerSimpleDAO"**

**class**=**"com.java2s.customer.dao.impl.SimpleJdbcCustomerDAO"**>

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

</bean>

<bean id=**"dataSource"**

**class**=**"org.springframework.jdbc.datasource.DriverManagerDataSource"**>

<property name=**"driverClassName"** value=**"${jdbc.driverClassName}"** />

<property name=**"url"** value=**"${jdbc.url}"** />

<property name=**"username"** value=**"${jdbc.username}"** />

<property name=**"password"** value=**"${jdbc.password}"** />

</bean>

</beans>

# Spring Tutorial - Spring Bean Inheritance

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0170__Spring_PlaceHolder_Properties.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0190__Spring_Dependency_Check.htm)

Spring supports bean configuration inheritance. We can define a bean and then further configure it to create new bean.

By using the bean inheritance we can share common values, properties or configurations.

A child bean inherits its parent bean configurations, properties and attributes.

In addition, the child beans can override the inherited value.

## Java Bean

**package** com.java2s.common;

**public** **class** Customer {

**private** **int** type;

**private** String action;

**private** String Country;

**public** **int** getType() {

**return** type;

}/\* w w w. ja v a2 s. c o m\*/

**public** **void** setType(**int** type) {

this.type = type;

}

**public** String getAction() {

**return** action;

}

**public** **void** setAction(String action) {

this.action = action;

}

**public** String getCountry() {

**return** Country;

}

**public** **void** setCountry(String country) {

Country = country;

}

@Override

**public** String toString() {

**return** **"Customer [type="** + type + **", action="** + action + **", Country="**

+ Country + **"]"**;

}

}

## Bean configuration file

Here is the Bean configuration file

<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-2.5.xsd">

<bean id=**"generalCustomer"** **class**=**"com.java2s.common.Customer"**>

<property name=**"country"** value=**"USA"** />

</bean>

<bean id=**"specialCustomer"** parent=**"generalCustomer"**>

<property name=**"action"** value=**"backup"** />

<property name=**"type"** value=**"1"** />

</bean>

</beans>

In the configuration file above 'generalCustomer' bean contains a 'USA' value for country property.

After the definition of 'generalCustomer' bean we defined the 'specialCustomer' bean inherited this value from its parent ('generalCustomer').

Here is the code to run this application.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Customer cust = (Customer)context.getBean(**"specialCustomer"**);

System.out.println(cust);

}

}

The code above generates the following result.

Customer [type=1, action=backup, Country=USA]

[Download Java2s\_Spring\_Bean\_Inheritance.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Bean_Inheritance.zip)

## abstract bean

In above example, we can still instantiate the 'generalCustomer' bean as follows.

Customer cust = (Customer)context.getBean(**"generalCustomer"**);

To make this base bean as a template and not allow Java code to instantiate it, add an 'abstract' attribute in the <bean> element.

<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-2.5.xsd">

<bean id=**"generalCustomer"** **class**=**"com.java2s.common.Customer"** **abstract**=**"true"**>

<property name=**"country"** value=**"USA"** />

</bean>

<bean id=**"specialCustomer"** parent=**"generalCustomer"**>

<property name=**"action"** value=**"backup"** />

<property name=**"type"** value=**"1"** />

</bean>

</beans>

After marking the 'generalCustomer' bean as abstract in the bean definition. It becomes a pure template for bean to inherit.

If we try to instantiate it, we will encounter the following error message.

Customer cust = (Customer)context.getBean(**"generalCustomer"**);

org.springframework.beans.factory.BeanIsAbstractException:

Error creating bean with name **'generalCustomer'**:

Bean definition is **abstract**

## Bean Template

A parent bean doesn't need class attribute, it just need common properties for sharing.

Here's is an example

<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-2.5.xsd">

<bean id=**"generalCustomer"** **abstract**=**"true"**>

<property name=**"country"** value=**"USA"** />

</bean>

<bean id=**"specialCustomer"** parent=**"generalCustomer"**

**class**=**"com.java2s.common.Customer"**>

<property name=**"action"** value=**"backup"** />

<property name=**"type"** value=**"1"** />

</bean>

</beans>

In the xml code above the generalCustomer bean is a template which hosts common properties to other bean to inherit.

## Overrride

We can override the inherited value by specify the new value in the child bean.

<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-2.5.xsd">

<bean id=**"generalCustomer"** **class**=**"com.java2s.common.Customer"** **abstract**=**"true"**>

<property name=**"country"** value=**"USA"** />

</bean>

<bean id=**"specialCustomer"** parent=**"generalCustomer"**>

<property name=**"country"** value=**"Japan"** />

<property name=**"action"** value=**"backup"** />

<property name=**"type"** value=**"1"** />

</bean>

</beans>

The 'specialCustomer' bean overrides the parent 'generalCustomer' country property, from 'USA' to 'Japan'.

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/O/OVERRRIDE__D78E6AAEDFB5010FB1DB.PNG   
  
[Download Java2s\_Spring\_Override.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Override.zip)

# Spring Tutorial - Spring Dependency Check

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0180__Spring_Bean_Inheritance.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0200__Spring_Bean_Initialize_Destroy.htm)

Spring can do dependency checking to make sure the required properties have been set or injected.

Spring supports four dependency checking modes.

* none - No dependency checking. Default mode.
* simple - If any properties of primitive type (int, long, double...) and collection types (map, list...) have not been set, Spring throws UnsatisfiedDependencyException.
* objects - If any properties of object type have not been set, Spring will throw UnsatisfiedDependencyException.
* all - If any properties of any type have not been set, Spring will throw an UnsatisfiedDependencyException.

## Java Bean

The following sections uses the two Java bean as follows to shows how to use the dependency checking.

Customer class.

package com.java2s.common;

**public** **class** Customer {

**private** Person person;

**private** int type;

**private** String action;

**public** Person getPerson() {

return person;

}

**public** void setPerson(Person person) {

this.person = person;

}

**public** int getType() {

return type;

}

**public** void setType(int type) {

this.type = type;

}

**public** String getAction() {

return action;

}

**public** void setAction(String action) {

this.action = action;

}

@Override

**public** String toString() {

return **"Customer [person="** + person + **", type="** + type + **", action="**

+ action + **"]"**;

}

}

Person class

package com.java2s.common;

**public** **class** Person

{

**private** String name;

**private** String address;

**private** int age;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

@Override

**public** String toString() {

return **"Person [name="** + name + **", address="** + address + **", age="** + age

+ **"]"**;

}

}

## dependency checking:none

The following code shows how to use spring bean configuration file with 'none' dependency checking mode.

<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-2.5.xsd">

<bean id=**"myCustomer"** **class**=**"com.java2s.common.Customer"** >

<property name=**"action"** value=**"buy"** />

</bean>

<bean id=**"myPerson"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

<property name=**"age"** value=**"29"** />

</bean>

</beans>

The default value to dependency checking is none. If you did not explicitly define the dependency checking mode, it's default to 'none'. Therefore no dependency checking will perform.

## dependency checking:simple

The following code shows how to use Spring bean configuration file with 'simple' dependency checking mode.

<bean id=**"myCustomer"** **class**=**"com.java2s.common.Customer"**

dependency-check=**"simple"**>

Full configuration file.

<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-2.5.xsd">

<bean id=**"myCustomer"** **class**=**"com.java2s.common.Customer"**

dependency-check=**"simple"**>

<property name=**"person"** ref=**"myPerson"** />

<property name=**"action"** value=**"buy"** />

</bean>

<bean id=**"myPerson"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

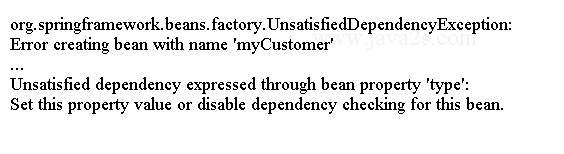
<property name=**"age"** value=**"29"** />

</bean>

</beans>

In the xml code above the myCustomer bean is marked to use simple dependency check. For the simple dependency check, if any properties of primitive type (int, long, double...) and collection types (map, list...) have not been set, Spring throws UnsatisfiedDependencyException.

The 'type' property is type int which is a primitive type has not been set, an UnsatisfiedDependencyException will throw as follows.



## Spring dependency checking with @Required Annotation

In the following we will introducing another way of doing dependency checking.

We can use @Required Annotation to add dependency checking for Java beans.

@Required annotation can apply to a particular property.

The following Customer object has @Required in setPerson() method to make sure the person property has been set.

package com.java2s.common;

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

**public** **class** Customer

{

**private** Person person;

**private** int type;

**private** String action;

**public** Person getPerson() {

return person;

}

@Required

**public** void setPerson(Person person) {

this.person = person;

}

**public** int getType() {

return type;

}

**public** void setType(int type) {

this.type = type;

}

**public** String getAction() {

return action;

}

**public** void setAction(String action) {

this.action = action;

}

}

After applying the @Required annotation on the method, we also need to register an RequiredAnnotationBeanPostProcessor to acknowledge the @Required annotation in bean configuration file.

There are two ways to enable the RequiredAnnotationBeanPostProcessor.

* Add Spring context and <context:annotation-config /> in bean configuration file.
* Include 'RequiredAnnotationBeanPostProcessor' directly in bean configuration file.

Here is the syntax of context:annotation-config.

<context:annotation-config />

Full source code,

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config />

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

<property name=**"age"** value=**"29"** />

</bean>

</beans>

The following xml code shows how to include 'RequiredAnnotationBeanPostProcessor' in bean configuration file.

<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-2.5.xsd">

<bean

**class**=**"org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"**/>

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

<bean id=**"PersonBean"** **class**=**"com.java2s.common.Person"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

<property name=**"age"** value=**"29"** />

</bean>

</beans>

If we run it , the following error message will be throw, because person property is unset.

org.springframework.beans.factory.BeanInitializationException:

Property **'person'** is required **for** bean **'CustomerBean'**

## Define Custom @Required-Style Annotation In Spring

We can define custom annotation to do dependency checking by using Spring @Required-style annotation.

In the following example, we will create a custom @Required-style annotation named @Mandatory, which is equivalent to @Required annotation.

First, we create the @Mandatory interface.

package com.java2s.common;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @**interface** Mandatory {

}

Then, we apply the new created annotation to a property from a Java Bean.

package com.java2s.common;

**public** **class** Customer

{

**private** Person person;

**private** int type;

**private** String action;

@Mandatory

**public** void setPerson(Person person) {

this.person = person;

}

}

Finally, we need to register it in the xml configuration file by including @Mandatory annotation in 'RequiredAnnotationBeanPostProcessor' class.

<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-2.5.xsd">

<bean **class**=**"org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"**>

<property name=**"requiredAnnotationType"** value=**"com.java2s.common.Mandatory"**/>

</bean>

<bean id=**"CustomerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

</beans>

# Spring Tutorial - Spring Bean Initialize and Destroy

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0190__Spring_Dependency_Check.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0300__Spring_Expression_Language.htm)

We can use the Spring InitializingBean and DisposableBean to perform actions on bean initialization and destruction.

For a bean implemented InitializingBean from Spring, the Spring IoC container will runafterPropertiesSet() method after all bean properties have been set.

For a bean implemented DisposableBean from Spring, it will call destroy() after Spring container is released the bean.

## Java Bean

Here's an example to show you how to use InitializingBean and DisposableBean.

A PrinterHelper bean is defined in the following code and it implements both InitializingBean and DisposableBean interface.

It also has two more methods one is afterPropertiesSet() method and the other isdestroy() method.

package com.java2s.common;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

**public** **class** PrinterHelper **implements** InitializingBean, DisposableBean

{

String message;

**public** String getMessage() {

return message;

}

**public** void setMessage(String message) {

this.message = message;

}

**public** void afterPropertiesSet() throws Exception {

System.out.println(**"Init method after properties are set : "** + message);

}

**public** void destroy() throws Exception {

System.out.println(**"Spring Container is destroy! JavaBean clean up"**);

}

@Override

**public** String toString() {

return message ;

}

}

## XML configuration file

Here is the Spring-Customer.xml for Java bean configuration.

<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-2.5.xsd">

<bean id=**"printerService"** **class**=**"com.java2s.common.PrinterHelper"**>

<property name=**"message"** value=**"i'm property message"** />

</bean>

</beans>

## Main method

Here is the code shows how to run the code above.

ConfigurableApplicationContext.close() call will close the application context and it will call destroy() method.

package com.java2s.common;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ConfigurableApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

PrinterHelper cust = (PrinterHelper)context.getBean(**"printerService"**);

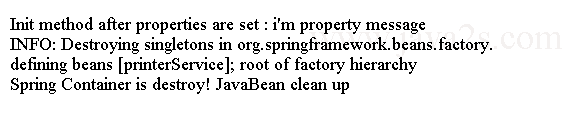
System.out.println(cust);

context.close();

}

}

The code above generates the following result.



From the output we can see that the afterPropertiesSet() method is called, after the message property is set and the destroy() method is call after the context.close();

[Download Java2s\_Spring\_Init\_Destroy.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Init_Destroy.zip)

## Spring init-method and destroy-method

The following code shows another way to call custom method in a Java bean when initializing and destroying a Java Bean.

We can use init-method and destroy-method attribute in bean configuration file to mark init method and destroy method in Java Bean.

The init-method is called during initialization and the destroy-method is called during destruction.

The following code shows how to add methods in a Java Bean to do init and destroy respectively.

package com.java2s.common;

**public** **class** PrinterHelper {

String message;

**public** String getMessage() {

return message;

}

**public** void setMessage(String message) {

this.message = message;

}

**public** void initIt() throws Exception {

System.out.println(**"Init method after properties are set : "** + message);

}

**public** void cleanUp() throws Exception {

System.out.println(**"Spring Container is destroy! Customer clean up"**);

}

@Override

**public** String toString() {

return message;

}

}

In the following xml configuration file it uses init-method and destroy-method attributes to mark the init method and destroy method repectively.

<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-2.5.xsd">

<bean id=**"printerService"** **class**=**"com.java2s.common.PrinterHelper"**

init-method=**"initIt"** destroy-method=**"cleanUp"**>

<property name=**"message"** value=**"i'm property message"** />

</bean>

</beans>

[Download Java2s\_Spring\_init-method.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_init-method.zip)

## @PostConstruct and @PreDestroy annotation

Spring also support the @PostConstruct and @PreDestroy annotation marked Java Bean.

We use those two annotations to mark init-method and destroy-method.

@PostConstruct and @PreDestroy annotation are from the J2ee common-annotations.jar.

In the following code we use @PostConstruct and @PreDestroy annotations to mark the PrinterHelper class.

package com.java2s.customer.services;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

**public** **class** PrinterHelper

{

String message;

**public** String getMessage() {

return message;

}

**public** void setMessage(String message) {

this.message = message;

}

@PostConstruct

**public** void initIt() throws Exception {

System.out.println(**"Init method after properties are set : "** + message);

}

@PreDestroy

**public** void cleanUp() throws Exception {

System.out.println(**"Spring Container is destroy! Customer clean up"**);

}

}

In order to use @PostConstruct and @PreDestroy annotations we need to either register 'CommonAnnotationBeanPostProcessor' or specify the '<context:annotation-config />' in bean configuration file.

The following xml configuraiton file calls CommonAnnotationBeanPostProcessor Java Bean.

<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-2.5.xsd">

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

<bean id=**"printerService"** **class**=**"com.java2s.common.PrinterHelper"**>

<property name=**"message"** value=**"i'm property message"** />

</bean>

</beans>

The following xml file calls <context:annotation-config />.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config />

<bean id=**"printerService"** **class**=**"com.java2s.common.PrinterHelper"**>

<property name=**"message"** value=**"i'm property message"** />

</bean>

</beans>

# Spring Tutorial - Spring Expression Language

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0200__Spring_Bean_Initialize_Destroy.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0310__Spring_Expression_Language_Reference.htm)

Spring 3.0 introduced a powerful expression language known as Spring expression language, or Spring EL.

The Spring Expression Language, available via XML or annotation, is evaluated or executed during the bean creation time.

## Spring expression language hello world

In the following code shows how to use Spring Expression Language to inject String, integer and bean into property, both in XML and annotation.

In order to use Spring Expression Language we need to add the following jars dependency to the pom.xml file.

...

<properties>

<spring.version>3.0.5.RELEASE</spring.version>

</properties>

<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>

...

## Java Bean

The following code defines two Java beans, later we will use Spring Expression Language to inject values into property in XML and annotation.

Server Java Bean.

package com.java2s.common;

**public** **class** Server {

**private** Item item;

**private** String itemName;

**public** Item getItem() {

return item;

}

**public** void setItem(Item item) {

this.item = item;

}

**public** String getItemName() {

return itemName;

}

**public** void setItemName(String itemName) {

this.itemName = itemName;

}

@Override

**public** String toString() {

return **"Server [item="** + item + **", itemName="** + itemName + **"]"**;

}

}

Item Java Bean.

package com.java2s.common;

**public** **class** Item {

**private** String name;

**private** int qty;

**public** String getName() {

return name;

}

@Override

**public** String toString() {

return **"Item [name="** + name + **", qty="** + qty + **"]"**;

}

**public** int getQty() {

return qty;

}

**public** void setQty(int qty) {

this.qty = qty;

}

**public** void setName(String name) {

this.name = name;

}

}

## Spring EL in XML

The following code shows how to use Spring EL in XML.

First, we defined a Java Bean Item in xml and set its property value. We set the name to have itemA value and set 10 as the qty value.

<bean id=**"itemBean"** **class**=**"com.java2s.common.Item"**>

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

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

</bean>

Then, we created the Server bean in XML by reusing the value from Item Bean.

The Spring Expression Language is enclosed with #{ expression }. The following code references the value from Item bean. It assigns the itemBean to item and itemBean.name to itemName. Value of itemBean.name is itemA.

<bean id=**"myServer"** **class**=**"com.java2s.common.Server"**>

<property name=**"item"** value=**"#{itemBean}"** />

<property name=**"itemName"** value=**"#{itemBean.name}"** />

</bean>

Full configuration file.

<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-3.0.xsd">

<bean id=**"itemBean"** **class**=**"com.java2s.common.Item"**>

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

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

</bean>

<bean id=**"myServer"** **class**=**"com.java2s.common.Server"**>

<property name=**"item"** value=**"#{itemBean}"** />

<property name=**"itemName"** value=**"#{itemBean.name}"** />

</bean>

</beans>

[Download Java2s\_Spring\_EL\_XML.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_EL_XML.zip)

## Spring EL in Annotation

The following example shows how to use Spring Expression Language in annotation.

First, we define a Java Bean Item and mark it with Component annotation. For its properties we use the Value annotation to assign them values

package com.java2s.common;

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

import org.springframework.stereotype.Component;

@Component(**"itemBean"**)

**public** **class** Item {

@Value(**"itemA"**) //inject String directly

**private** String name;

@Value(**"10"**) //inject interger directly

**private** int qty;

**public** String getName() {

return name;

}

@Override

**public** String toString() {

return **"Item [name="** + name + **", qty="** + qty + **"]"**;

}

**public** int getQty() {

return qty;

}

**public** void setQty(int qty) {

this.qty = qty;

}

**public** void setName(String name) {

this.name = name;

}

}

Then, we define a Server Java bean and also mark it with Component annotation. When defining Server's properties we use the values defined Value annotations from Item bean.

package com.java2s.common;

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

import org.springframework.stereotype.Component;

@Component(**"myServer"**)

**public** **class** Server {

@Value(**"#{itemBean}"**)

**private** Item item;

@Value(**"#{itemBean.name}"**)

**private** String itemName;

**public** Item getItem() {

return item;

}

**public** void setItem(Item item) {

this.item = item;

}

**public** String getItemName() {

return itemName;

}

**public** void setItemName(String itemName) {

this.itemName = itemName;

}

@Override

**public** String toString() {

return **"Server [item="** + item + **", itemName="** + itemName + **"]"**;

}

}

Finally, we have to enable auto component scanning in the xml file.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

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

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

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

<context:component-scan base-package=**"com.java2s.common"** />

</beans>

## Example

The following code shows how to run the code above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Server obj = (Server) context.getBean(**"myServer"**);

System.out.println(obj);

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/E/EXAMPLE__C117FA70EF9F800EC336.PNG   
  
[Download Java2s\_spring\_EL\_Annotation.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_spring_EL_Annotation.zip)

## Collections in Spring Expression Language

The following code shows how to use Collections in Spring Expression Language.

First, define a Java bean with collections in it.

package com.java2s.common;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.springframework.stereotype.Component;

@Component(**"testBean"**)

**public** **class** Test {

**private** Map<String, String> map;

**private** List<String> list;

**public** Test() {

map = **new** HashMap<String, String>();

map.put(**"MapA"**, **"This is A"**);

map.put(**"MapB"**, **"This is B"**);

map.put(**"MapC"**, **"This is C"**);

list = **new** ArrayList<String>();

list.add(**"List0"**);

list.add(**"List1"**);

list.add(**"List2"**);

}

}

Then, use the collection in the expression language.

package com.java2s.common;

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

import org.springframework.stereotype.Component;

@Component(**"customerBean"**)

**public** **class** Customer {

@Value(**"#{testBean.map['MapA']}"**)

**private** String mapA;

@Value(**"#{testBean.list[0]}"**)

**private** String list;

}

Use the same settings in the xml configuration file.

<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-3.0.xsd">

<bean id=**"customerBean"** **class**=**"com.java2s.common.Customer"**>

<property name=**"mapA"** value=**"#{testBean.map['MapA']}"** />

<property name=**"list"** value=**"#{testBean.list[0]}"** />

</bean>

<bean id=**"testBean"** **class**=**"com.java2s.common.Test"** />

</beans>

# Spring Tutorial - Spring Expression Language Usage

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0300__Spring_Expression_Language.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0320__Spring_Expression_Language_Operator.htm)

In Spring Expression Language, we can reference a bean and its nested properties using a 'dot (.)' syntax.

bean.property\_name

In the following code, we inject the value of "country" property from "addressBean" bean into "customer" class "country" property.

**public** **class** Server {

@Value(**"#{addressBean.country}"**)

**private** String country;

...

}

## Example

The following code defines an Address bean and marks the bean with Spring Expression Language.

It fills the street with string value, fills the postcode with int value. It also defines a utility method getFullAddress to return the full address from post code, street, and contry.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"addressBean"**)

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

@Value(**"Main Street, New York"**)

**private** String street;

@Value(**"123456"**)

**private** int postcode;

@Value(**"US"**)

**private** String country;

**public** String getFullAddress(String prefix) {

return prefix + **" : "** + street + **" "** + postcode + **" "** + country;

}

**public** void setCountry(String country) {

this.country = country;

}

@Override

**public** String toString() {

return **"Address [street="** + street + **", postcode="** + postcode

+ **", country="** + country + **"]"**;

}

**public** String getStreet() {

return street;

}

**public** void setStreet(String street) {

this.street = street;

}

**public** int getPostcode() {

return postcode;

}

**public** void setPostcode(int postcode) {

this.postcode = postcode;

}

**public** String getCountry() {

return country;

}

}

The following code uses the value defined in the Address Java bean to fill the properties in the Server bean.

In Spring Expression language we can also call the method from a bean.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"myServer"**)

**public** **class** Server {

@Value(**"#{addressBean}"**)

**private** Address address;

@Value(**"#{addressBean.country}"**)

**private** String country;

@Value(**"#{addressBean.getFullAddress('java2s')}"**)

**private** String fullAddress;

@Override

**public** String toString() {

return **"Server [address="** + address + **"\n, country="** + country

+ **"\n, fullAddress="** + fullAddress + **"]"**;

}

**public** Address getAddress() {

return address;

}

**public** void setAddress(Address address) {

this.address = address;

}

**public** String getCountry() {

return country;

}

**public** void setCountry(String country) {

this.country = country;

}

**public** String getFullAddress() {

return fullAddress;

}

**public** void setFullAddress(String fullAddress) {

this.fullAddress = fullAddress;

}

}

The following code shows how to fill the same data in xml file configuration.

<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-3.0.xsd">

<bean id=**"myServer"** **class**=**"com.java2s.core.Server"**>

<property name=**"address"** value=**"#{addressBean}"** />

<property name=**"country"** value=**"#{addressBean.country}"** />

<property name=**"fullAddress"** value=**"#{addressBean.getFullAddress('java2s')}"** />

</bean>

<bean id=**"addressBean"** **class**=**"com.java2s.core.Address"**>

<property name=**"street"** value=**"Main Street, New York"** />

<property name=**"postcode"** value=**"123456"** />

<property name=**"country"** value=**"US"** />

</bean>

</beans>

## Test

<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-3.0.xsd">

<bean id=**"myServer"** **class**=**"com.java2s.core.Server"**>

</bean>

</beans>

The following code shows how to run the code above.

package com.java2s.core;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

Server obj = (Server) context.getBean(**"myServer"**);

System.out.println(obj);

}

}

## Example 2

The following code shows how to call a method with no parameter in Spring Expression Language.

First, we define a Java bean with a method to return a double value.

package com.java2s.core;

import org.springframework.stereotype.Component;

@Component(**"priceBean"**)

**public** **class** Price {

**public** double getSpecialPrice() {

return **new** Double(99.99);

}

}

In the following code we call the method defined above in Spring Expression Language.

@Value(**"#{priceBean.getSpecialPrice()}"**)

**private** double amount;

We can also call the 'toUpperCase()' method on the String literal.

@Value(**"#{'java2s'.toUpperCase()}"**)

**private** String name;

Full source code

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"myServer"**)

**public** **class** Server {

@Value(**"#{'java2s'.toUpperCase()}"**)

**private** String name;

@Value(**"#{priceBean.getSpecialPrice()}"**)

**private** double amount;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** double getAmount() {

return amount;

}

**public** void setAmount(double amount) {

this.amount = amount;

}

@Override

**public** String toString() {

return **"Server [name="** + name + **", amount="** + amount + **"]"**;

}

}

The following code shows how to rewrite the code above in bean definition XML file.

<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-3.0.xsd">

<bean id=**"myServer"** **class**=**"com.java2s.core.Server"**>

<property name=**"name"** value=**"#{'java2s'.toUpperCase()}"** />

<property name=**"amount"** value=**"#{priceBean.getSpecialPrice()}"** />

</bean>

<bean id=**"priceBean"** **class**=**"com.java2s.core.Price"** />

</beans>

# Spring Tutorial - Spring Expression Language Operators

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0310__Spring_Expression_Language_Reference.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0330__Spring_Auto_Scan_Bean.htm)

Spring Expression Language supports the standard mathematical, logical or relational operators.

The following Relational operators are supported.

* equal: ==, eq
* not equal: !=, ne
* less than: <, lt
* less than or equal: <= , le
* greater than: >, gt
* greater than or equal: >=, ge

The following Logical operators are supported.

* and
* or
* not (!)

The following Mathematical operators are supported.

* addition (+)
* Subtraction (-)
* Multiplication (\*)
* Division (/)
* Modulus (%)
* Exponential power (^)

## Example

The following code shows how to use operators from Spring expression language.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"customerBean"**)

**public** **class** Server {

//Relational operators

@Value(**"#{1 == 1}"**) //true

**private** boolean testEqual;

@Value(**"#{1 != 1}"**) //false

**private** boolean testNotEqual;

@Value(**"#{1 < 1}"**) //false

**private** boolean testLessThan;

@Value(**"#{1 <= 1}"**) //true

**private** boolean testLessThanOrEqual;

@Value(**"#{1 > 1}"**) //false

**private** boolean testGreaterThan;

@Value(**"#{1 >= 1}"**) //true

**private** boolean testGreaterThanOrEqual;

//Logical operators , numberBean.no == 999

@Value(**"#{numberBean.no == 999 and numberBean.no < 900}"**) //false

**private** boolean testAnd;

@Value(**"#{numberBean.no == 999 or numberBean.no < 900}"**) //true

**private** boolean testOr;

@Value(**"#{!(numberBean.no == 999)}"**) //false

**private** boolean testNot;

//Mathematical operators

@Value(**"#{1 + 1}"**) //2.0

**private** double testAdd;

@Value(**"#{'1' + '@' + '1'}"**) //1@1

**private** String testAddString;

@Value(**"#{1 - 1}"**) //0.0

**private** double testSubtraction;

@Value(**"#{1 \* 1}"**) //1.0

**private** double testMultiplication;

@Value(**"#{10 / 2}"**) //5.0

**private** double testDivision;

@Value(**"#{10 % 10}"**) //0.0

**private** double testModulus ;

@Value(**"#{2 ^ 2}"**) //4.0

**private** double testExponentialPower;

}

## Spring EL Operator in XML

We can also use the operators in bean definition XML file.

<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-3.0.xsd">

<bean id=**"customerBean"** **class**=**"com.java2s.core.Customer"**>

<property name=**"testEqual"** value=**"#{1 == 1}"** />

<property name=**"testNotEqual"** value=**"#{1 != 1}"** />

<property name=**"testLessThan"** value=**"#{1 lt 1}"** />

<property name=**"testLessThanOrEqual"** value=**"#{1 le 1}"** />

<property name=**"testGreaterThan"** value=**"#{1 > 1}"** />

<property name=**"testGreaterThanOrEqual"** value=**"#{1 >= 1}"** />

<property name=**"testAnd"** value=**"#{numberBean.no == 999 and numberBean.no lt 900}"** />

<property name=**"testOr"** value=**"#{numberBean.no == 999 or numberBean.no lt 900}"** />

<property name=**"testNot"** value=**"#{!(numberBean.no == 999)}"** />

<property name=**"testAdd"** value=**"#{1 + 1}"** />

<property name=**"testAddString"** value=**"#{'1' + '@' + '1'}"** />

<property name=**"testSubtraction"** value=**"#{1 - 1}"** />

<property name=**"testMultiplication"** value=**"#{1 \* 1}"** />

<property name=**"testDivision"** value=**"#{10 / 2}"** />

<property name=**"testModulus"** value=**"#{10 % 10}"** />

<property name=**"testExponentialPower"** value=**"#{2 ^ 2}"** />

</bean>

</beans>

## Spring EL ternary operator (if-then-else)

Spring Expression Language ternary operator has the following syntax. and it performs if then else conditional logic.

condition ? trueAction : falseAction

If condition is true it will execute the trueAction, if the the condition is false it will run the falseAction.

The following Java bean has a property value for qtyOnHand whose value is 99.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"itemBean"**)

**public** **class** Item {

@Value(**"99"**)

**private** int qtyOnHand;

}

The Customer bean use ternary operator with @Value annotation. If "itemBean.qtyOnHand" is less than 100, then set "customerBean.warning" to true, else set it to false.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"customerBean"**)

**public** **class** Customer {

@Value(**"#{itemBean.qtyOnHand < 100 ? true : false}"**)

**private** boolean warning;

}

The following xml configuration file shows how to use tenary operator in xml markup.

<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-3.0.xsd">

<bean id=**"itemBean"** **class**=**"com.java2s.core.Item"**>

<property name=**"qtyOnHand"** value=**"99"** />

</bean>

<bean id=**"customerBean"** **class**=**"com.java2s.core.Customer"**>

<property name=**"warning"**

value=**"#{itemBean.qtyOnHand &lt; 100 ? true : false}"** />

</bean>

</beans>

## Spring EL regular expression

The following Java bean has an email field which will be validated by using regular expression.

package com.java2s.core;

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

import org.springframework.stereotype.Component;

@Component(**"emailBean"**)

**public** **class** Email {

@Value(**"your email here"**)

String emailAddress;

}

The following codes use regular expression to validate a number and stores the result in boolean value.

// **if** this is a digit?

@Value(**"#{'1' matches '\\d+' }"**)

**private** boolean validDigit;

The following code uses the regular expression in tenary operator.

@Value(**"#{ ('abc' matches '\\d+') == true ? "** +

**"'yes this is digit' : 'No this is not a digit' }"**)

**private** String msg;

The following code uses regular expression to validate an email address from another Java bean.

// email regular expression

String emailRegEx = **"^[\_A-Za-z0-9-]+(\\.[\_A-Za-z0-9-]+)"** +

**"\*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)\*(\\.[A-Za-z]{2,})$"**;

@Value(**"#{emailBean.emailAddress matches customerBean.emailRegEx}"**)

**private** boolean validEmail;

Use the same regular expression in xml

...

<bean id=**"customerBean"** **class**=**"com.java2s.core.Customer"**>

<property name=**"validDigit"** value=**"#{'1' matches '\d+' }"** />

<property name=**"msg"**

value=**"#{ ('abc' matches '\d+') == true ? 'yes this is digit' : 'No this is not a digit' }"** />

<property name=**"validEmail"**

value=**"#{emailBean.emailAddress matches '^[\_A-Za-z0-9-]+(\.[\_A-Za-z0-9-]+)\*@[A-Za-z0-9]+(\.[A-Za-z0-9]+)\*(\.[A-Za-z]{2,})$' }"** />

</bean>

<bean id=**"emailBean"** **class**=**"com.java2s.core.Email"**>

<property name=**"emailAddress"** value=**"your email"** />

</bean>

</beans>

# Spring Tutorial - Spring Bean Auto Scan

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0320__Spring_Expression_Language_Operator.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0330__Spring_Auto_Wire_Bean.htm)

Spring can auto scan, detect, wire and instantiate Java beans.

The following sections shows the difference between xml configuration vs auto scan auto wire configuration.

## Old fashion

Here is a normal Java bean.

package com.java2s.common;

**public** **class** Printer

{

}

Another Java bean with referencing the code above.

package com.java2s.common;

**public** **class** PrinterHelper

{

Printer printer;

**public** void setPrinter(Printer myPrinter) {

this.printer = myPrinter;

}

@Override

**public** String toString() {

return **"PrinterHelper [myPrinter="** + printer + **"]"**;

}

}

XML Bean configuration file Spring-Customer.xml with a normal bean configuration in Spring.

<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-2.5.xsd">

<bean id=**"printerHelper"** **class**=**"com.java2s.common.PrinterHelper"**>

<property name=**"printer"** ref=**"myPrinter"** />

</bean>

<bean id=**"myPrinter"** **class**=**"com.java2s.common.Printer"** />

</beans>

Here is the code to run.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

PrinterHelper cust = (PrinterHelper)context.getBean(**"printerHelper"**);

System.out.println(cust);

}

}

[Download Java2s\_Spring\_Not\_Autowired.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Not_Autowired.zip)

## Auto Java bean Scan and wire

To enable Spring auto component scanning features, add annotation @Component to class.

package com.java2s.common;

import org.springframework.stereotype.Component;

@Component

**public** **class** Printer

{

}

The following Java bean uses not only @Component to indicate this is an auto scan component, it also marks the property as Autowired.

package com.java2s.common;

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

import org.springframework.stereotype.Component;

@Component

**public** **class** PrinterHelper

{

@Autowired

Printer myPrinter;

}

The following xml configuration file finally adds the package to enable the auto wire auto scan context.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package=**"com.java2s.common"** />

</beans>

[Download Java2s\_Spring\_Autowire.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Autowire.zip)

## Note

By default, Spring will lower case the first character of the component's simple class name as the bean id.

The following code shows how to use the auto-generated id to retrieve the component.

PrinterHelper cust = (PrinterHelper)context.getBean(**"printerHelper"**);

We can also name a component when declaring it.

@Component(**"myService"**)

**public** **class** PrinterHelper {

}

Once we define the name in @Component, we can retrieve it with this name 'myService'.

PrinterHelper cust = (PrinterHelper)context.getBean(**"myService"**);

## Filter to include

The following code shows how to use Spring filter to scan and register components by regular expressions, even the class is not annotated with @Component.

package com.java2s.common;

**public** **class** Printer

{

}

Helper class

package com.java2s.common;

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

**public** **class** PrinterHelper

{

@Autowired

Printer myPrinter;

}

The following xml configuration uses the Spring filter to include classes.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package=**"com.java2s"** >

<context:include-filter type=**"regex"**

expression=**"com.java2s.common.\*DAO.\*"** />

<context:include-filter type=**"regex"**

expression=**"com.java2s.common.\*Service.\*"** />

</context:component-scan>

</beans>

The following code includes any classes under com.java2s.customer.dao package with DAO in class name.

<context:include-filter type=**"regex"**

expression=**"com.java2s.common.\*DAO.\*"** />

Run it

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.java2s.customer.services.PrinterHelper;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

PrinterHelper cust = (PrinterHelper)context.getBean(**"printerHelper"**);

System.out.println(cust);

}

}

## Filter to exclude

The following code shows how to exclude specified components to stop Spring from auto registering for classes annotated with @Service.

<context:component-scan base-package=**"com.java2s.common"** >

<context:exclude-filter type=**"annotation"**

expression=**"org.springframework.stereotype.Service"** />

</context:component-scan>

Exclude classes with DAO letters in class name.

<context:component-scan base-package=**"com.java2s"** >

<context:exclude-filter type=**"regex"**

expression=**"com.java2s.customer.dao.\*DAO.\*"** />

</context:component-scan>

# Spring Tutorial - Spring Auto Wire Bean

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0330__Spring_Auto_Scan_Bean.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0340__Spring_Auto_Wire_Mode.htm)

Spring can use @Autowired annotation to auto wire bean on the setter method, constructor or a field.

## Java Beans

The following code defines two Java beans.

package com.java2s.common;

**public** **class** JobTitle

{

@Override

**public** String toString() {

return **"JobTitle [person="** + person + **", type="** + type + **", action="**

+ action + **"]"**;

}

**public** Employee getPerson() {

return person;

}

**public** void setPerson(Employee person) {

this.person = person;

}

**public** int getType() {

return type;

}

**public** void setType(int type) {

this.type = type;

}

**public** String getAction() {

return action;

}

**public** void setAction(String action) {

this.action = action;

}

**private** Employee person;

**private** int type;

**private** String action;

}

Employee Bean

package com.java2s.common;

**public** **class** Employee{

**private** String name;

**private** String address;

**private** int age;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

@Override

**public** String toString() {

return **"Employee [name="** + name + **", address="** + address + **", age="**

+ age + **"]"**;

}

}

## enable @Autowired

To enable @Autowired, we have to register 'AutowiredAnnotationBeanPostProcessor'. And we can do that in two ways.

* Add Spring context and <context:annotation-config /> in bean configuration file.
* Include 'AutowiredAnnotationBeanPostProcessor' in bean configuration file.

Here is the code to Add Spring context and <context:annotation-config />.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config />

<bean id=**"myJobTitle"** **class**=**"com.java2s.common.JobTitle"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

<bean id=**"myEmp"** **class**=**"com.java2s.common.Employee"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

<property name=**"age"** value=**"29"** />

</bean>

</beans>

The following code shows how to include 'AutowiredAnnotationBeanPostProcessor' in bean configuration file.

<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-2.5.xsd">

<bean

**class**=**"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"**/>

<bean id=**"myJobTitle"** **class**=**"com.java2s.common.JobTitle"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

<bean id=**"myEmp"** **class**=**"com.java2s.common.Employee"**>

<property name=**"name"** value=**"java2s"** />

<property name=**"address"** value=**"address ABC"** />

<property name=**"age"** value=**"29"** />

</bean>

</beans>

Here is the code to run.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

JobTitle cust = (JobTitle)context.getBean(**"myJobTitle"**);

System.out.println(cust);

}

}

[Download Java2s\_Spring\_autowire\_1.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_autowire_1.zip)

## @Autowired Examples

The following code shows how to autowire bean via @Autowired on setter method

package com.java2s.common;

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

**public** **class** JobTitle

{

@Override

**public** String toString() {

return **"JobTitle [person="** + person + **", type="** + type + **", action="**

+ action + **"]"**;

}

**public** Employee getPerson() {

return person;

}

@Autowired

**public** void setPerson(Employee person) {

this.person = person;

}

**public** int getType() {

return type;

}

**public** void setType(int type) {

this.type = type;

}

**public** String getAction() {

return action;

}

**public** void setAction(String action) {

this.action = action;

}

**private** Employee person;

**private** int type;

**private** String action;

}

The following code shows how to autowire bean via @Autowired on constructor.

package com.java2s.common;

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

**public** **class** JobTitle

{

@Autowired

**public** JobTitle(Employee person) {

this.person = person;

}

@Override

**public** String toString() {

return **"JobTitle [person="** + person + **", type="** + type + **", action="**

+ action + **"]"**;

}

**public** Employee getPerson() {

return person;

}

**public** void setPerson(Employee person) {

this.person = person;

}

**public** int getType() {

return type;

}

**public** void setType(int type) {

this.type = type;

}

**public** String getAction() {

return action;

}

**public** void setAction(String action) {

this.action = action;

}

**private** Employee person;

**private** int type;

**private** String action;

}

The following code shows how to autowire bean via @Autowired on a field.

package com.java2s.common;

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

**public** **class** JobTitle

{

@Autowired

**private** Employee person;

**private** int type;

**private** String action;

}

All of the above three methods autowired Employee Java bean into JobTitle's person property.

## Example

Here is the code to run the code above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App

{

**public** **static** void main( String[] args )

{

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {**"SpringBeans.xml"**});

JobTitle cust = (JobTitle)context.getBean(**"myJobTitle"**);

System.out.println(cust);

}

}

Output:

http://www.java2s.com/Tutorials/JavaImage/myResult/E/EXAMPLE__0AC2846A916CE44E669A.PNG   
  
[Download Java2s\_Spring\_Autowire\_Setter.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Autowire_Setter.zip)

## Dependency checking

By default @Autowired annotation performs the dependency checking to ensure the autowired bean exist.

If Spring can't find a matching bean, it will throw an exception. To disable @Autowired dependency check, set the "required" attribute of @Autowired to false.

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

**public** **class** JobTitle

{

@Autowired(required=false)

**private** Employee person;

}

## Use @Qualifier to select bean

@Qualifier annotation allows us to select Java bean to do autowire on a field.

We need @Qualifier annotation when we have two qualified beans to auto wire.

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

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

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

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

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

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

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config />

<bean id=**"myJobTitle"** **class**=**"com.java2s.common.JobTitle"**>

<property name=**"action"** value=**"buy"** />

<property name=**"type"** value=**"1"** />

</bean>

<bean id=**"myEmp1"** **class**=**"com.java2s.common.Employee"**>

<property name=**"name"** value=**"java2s1"** />

<property name=**"address"** value=**"address 1"** />

<property name=**"age"** value=**"28"** />

</bean>

<bean id=**"myEmp2"** **class**=**"com.java2s.common.Employee"**>

<property name=**"name"** value=**"java2s2"** />

<property name=**"address"** value=**"address 2"** />

<property name=**"age"** value=**"28"** />

</bean>

</beans>

In the xml code configuration file above we have two instances of com.java2s.common.Employee. If we do not specify which one to autowire to JobTitle, they both qualified for the autowire action.

To select which Java bean to use we can use the @Qualifier annotation.

package com.java2s.common;

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

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

**public** **class** JobTitle

{

@Autowired

@Qualifier(**"myEmp1"**)

**private** Employee person;

}

# Spring Tutorial - Spring Auto Wire Bean

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0330__Spring_Auto_Wire_Bean.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0400__Spring_AOP.htm)

Spring can wire beans automatically. To enable it, define the "autowire" attribute in <bean>.

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"byName"** />

Spring has five Auto-wiring modes.

* no - Default, no auto wiring
* byName - Auto wiring by property name.
* byType - Auto wiring by property data type.
* constructor - byType mode in constructor argument.
* autodetect - If a default constructor is found, use "autowired by constructor"; Otherwise, use "autowire by type".

## Java Beans

Customer Java bean.

package com.java2s.common;

**public** **class** Customer

{

**private** Person person;

**public** Customer(Person person) {

this.person = person;

}

**public** void setPerson(Person person) {

this.person = person;

}

}

Person Java bean

package com.java2s.common;

**public** **class** Person

{

}

## Auto-Wiring 'no'

This is the default mode, we need to wire the Java bean via 'ref' attribute.

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"**>

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

</bean>

<bean id=**"person"** **class**=**"com.java2s.common.Person"** />

## Auto-Wiring 'byName'

The following code add autowire byName to the bean declaration.

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"byName"** />

Since the name of "person" bean is same with the name of the "customer" bean's "person" property, Spring will auto wire it via method setPerson(Person person).

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"byName"** />

<bean id=**"person"** **class**=**"com.java2s.common.Person"** />

## Auto-Wiring 'byType'

The following xml configuration declares the autowire type as byType.

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"byType"** />

Since the data type of "person" bean is same as the data type of the "customer" bean's property person object, Spring will auto wire it via method setPerson(Person person).

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"byType"** />

<bean id=**"person"** **class**=**"com.java2s.common.Person"** />

## Auto-Wiring 'constructor'

The following code declares a bean's autowire type as constructor.

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"constructor"** />

The data type of "person" bean is same as the constructor argument data type in "customer" bean's property (Person object), Spring will auto wire it via constructor method - "public Customer(Person person)".

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"constructor"** />

<bean id=**"person"** **class**=**"com.java2s.common.Person"** />

## Auto-Wiring 'autodetect'

The following code shows how to use autodetect autowire. If a constructor is found, uses "constructor"; Otherwise, uses "byType".

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"**

autowire=**"autodetect"** dependency-check="objects />

Since there is a constructor in "Customer" class, Spring will auto wire it via constructor method - "public Customer(Person person)".

<bean id=**"customer"** **class**=**"com.java2s.common.Customer"** autowire=**"autodetect"** />

<bean id=**"person"** **class**=**"com.java2s.common.Person"** />

# Spring Tutorial - Spring AOP

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0340__Spring_Auto_Wire_Mode.htm)
* [Next »](http://www.java2s.com/Tutorials/Java/Spring/0500__Spring_Object_XML_Mapping.htm)

Besides Dependency Injection (DI), another core feature that the Spring Framework offers is aspect-oriented programming (AOP).

AOP is a tool for implementing crosscutting concerns.

The crosscutting concerns refers to logic in an application that cannot be decomposed from the rest of the application and may result in code duplication and tight coupling.

AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

Adding Log methods to every single method is a cross-cutting concern. A logging strategy affects every logged part of the system. The log methods would cross-cut all logged classes and methods.

Spring AOP framework modularizes cross-cutting concerns in aspects. When executing a method in Spring IoC container, Spring AOP can hijack the executing method, and add extra functionality before or after the method execution.

## AOP Concepts

AOP comes with its own specific set of concepts and terms.

The following are the core concepts of AOP:

* Joinpoints is a point during the execution of your application. Typical example of joinpoints is a call to a method. Joinpoints define the points in your application at which you can insert additional logic using AOP.
* Advice is the code executed at a particular joinpoint. There are many types of advice, such as before , which executes before the joinpoint, and after , which executes after it.
* Pointcut is a collection of joinpoints that is used to define when advice should be executed. By creating pointcuts, we control over how to apply advice to the Java Bean in the application.
* Weaving is the process of inserting aspects into the application code at the appropriate point.
* An object whose execution flow is modified by an AOP process is referred to as the target object or advised object.

Spring AOP has four type of advices.

* Before advice - Run before the method execution
* After returning advice - Run after the method returns a result
* After throwing advice - Run after the method throws an exception
* Around advice - Run around the method execution, combine all three advices above.

Advice is an action to take either before or after the method execution.

## dependency

Add the following new dependency for AOP coding to POM.xml.

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>2.2.2</version>

</dependency>

## Java Bean

The following code defines a Java bean for a printer service.

package com.java2s.common;

**public** **class** PrinterService {

**private** String name;

**private** String url;

**public** void setName(String name) {

this.name = name;

}

**public** void setUrl(String url) {

this.url = url;

}

**public** void printName() {

System.out.println(**"Printer Name : "** + this.name);

}

**public** void printURL() {

System.out.println(**"Printer URL : "** + this.url);

}

**public** void printThrowException() {

**throw** **new** IllegalArgumentException();

}

}

## XML configuration

Here is the Spring-Customer.xml file for bean configuration.

<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-2.5.xsd">

<bean id=**"myService"** **class**=**"com.java2s.common.PrinterService"**>

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

<property name=**"url"** value=**"http://www.java2s.com"** />

</bean>

</beans>

The following is the main class to run the code above.

package com.java2s.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** void main(String[] args) {

ApplicationContext appContext = **new** ClassPathXmlApplicationContext(

**new** String[] { **"SpringBeans.xml"** });

PrinterService cust = (PrinterService) appContext.getBean(**"myService"**);

cust.printName();

cust.printURL();

**try** {

cust.printThrowException();

} **catch** (Exception e) {

}

}

}

Output

   
  
[Download Java2s\_Spring\_Pre\_AOP.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Java2s_Spring_Pre_AOP.zip)

## Before advice

The following code shows how to add Before advice.

A Before advice is executed before the method execution.

First, create a class which implements MethodBeforeAdvice interface.

package com.java2s.common;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

**public** **class** AOPBeforeMethod **implements** MethodBeforeAdvice

{

@Override

**public** void before(Method method, Object[] args, Object target)

throws Throwable {

System.out.println(**"AOPBeforeMethod : Before method call."**);

}

}

Then, we create the AOPBeforeMethod bean in the xml configuration file.

<bean id=**"aopBeforeMethodBean"** **class**=**"com.java2s.aop.AOPBeforeMethod"** />

To use the AOPBeforeMethod class we have to install it by usingorg.springframework.aop.framework.ProxyFactoryBean as follows.

<bean id=**"myServiceProxy"**

**class**=**"org.springframework.aop.framework.ProxyFactoryBean"**>

<property name=**"target"** ref=**"myService"** />

<property name=**"interceptorNames"**>

<list>

<value>aopBeforeMethodBean</value>

</list>

</property>

</bean>

Full source code

<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-2.5.xsd">

<bean id=**"myService"** **class**=**"com.java2s.common.PrinterService"**>

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

<property name=**"url"** value=**"http://www.java2s.com"** />

</bean>

<bean id=**"aopBeforeMethodBean"** **class**=**"com.java2s.common.AOPBeforeMethod"** />

<bean id=**"myServiceProxy"**

**class**=**"org.springframework.aop.framework.ProxyFactoryBean"**>

<property name=**"target"** ref=**"myService"** />

<property name=**"interceptorNames"**>

<list>

<value>aopBeforeMethodBean</value>

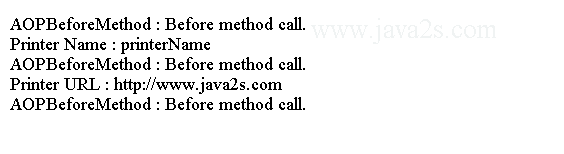
</list>

</property>

</bean>

</beans>

Run the main function again.



From the result we can see that the AOPBeforeMethod's before() method is executed before every myService's methods.

## After returning advice

The following code shows how to use After returning advice. After returning advice will execute after the method is returned a result.

First, create a class which implements AfterReturningAdvice interface.

package com.java2s.common;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

**public** **class** AOPAfterMethod **implements** AfterReturningAdvice

{

@Override

**public** void afterReturning(Object returnValue, Method method,

Object[] args, Object target) throws Throwable {

System.out.println(**"AOPAfterMethod : After method call."**);

}

}

Then, install the AOPAfterMethod class in the Bean configuration file.

<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-2.5.xsd">

<bean id=**"myService"** **class**=**"com.java2s.common.PrinterService"**>

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

<property name=**"url"** value=**"http://www.java2s.com"** />

</bean>

<bean id=**"aopAfterMethodBean"** **class**=**"com.java2s.common.AOPAfterMethod"** />

<bean id=**"myServiceProxy"**

**class**=**"org.springframework.aop.framework.ProxyFactoryBean"**>

<property name=**"target"** ref=**"myService"** />

<property name=**"interceptorNames"**>

<list>

<value>aopAfterMethodBean</value>

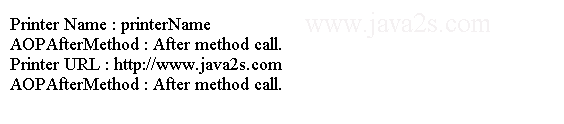
</list>

</property>

</bean>

</beans>

Run the main function again. Here is the output.



From the result we can see that the AOPAfterMethod's afterReturning() method is executed after every myService's methods returns a result.

## After throwing advice

The following code shows how to use After throwing advice. After throwing advice will execute after the method throws an exception.

First, create a class which implements ThrowsAdvice interface, and create a afterThrowing method to hijack the IllegalArgumentException exception.

package com.java2s.common;

import org.springframework.aop.ThrowsAdvice;

**public** **class** AOPThrowException **implements** ThrowsAdvice {

**public** void afterThrowing(IllegalArgumentException e) throws Throwable {

System.out.println(**"AOPThrowException : Throw exception call."**);

}

}

Then, install the AOPThrowException in Bean configuration file.

<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-2.5.xsd">

<bean id=**"myService"** **class**=**"com.java2s.common.PrinterService"**>

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

<property name=**"url"** value=**"http://www.java2s.com"** />

</bean>

<bean id=**"aopThrowExceptionBean"** **class**=**"com.java2s.common.AOPThrowException"** />

<bean id=**"myServiceProxy"**

**class**=**"org.springframework.aop.framework.ProxyFactoryBean"**>

<property name=**"target"** ref=**"myService"** />

<property name=**"interceptorNames"**>

<list>

<value>aopThrowExceptionBean</value>

</list>

</property>

</bean>

</beans>

Run the code and here is the output.

Printer Name : printerName

Printer URL : http://www.java2s.com

AOPThrowException : Throw exception call.

From the result we can see that Spring IoC container runs the AOPThrowException's afterThrowing() method when myService's methods throws an exception.

## Around advice

The following code shows how to use Around advice. Around advice combines all three advices above, and execute during method execution.

First, create a class which implements MethodInterceptor interface.

public Object invoke(MethodInvocation methodInvocation) throws Throwable method is called for each method call and we have to call the "methodInvocation.proceed();" to run the original method, otherwise the original method will not run.

package com.java2s.common;

import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;

import org.aopalliance.intercept.MethodInvocation;

**public** **class** AOPAroundMethod **implements** MethodInterceptor {

@Override

**public** Object invoke(MethodInvocation methodInvocation) throws Throwable {

System.out.println(**"Method name : "**

+ methodInvocation.getMethod().getName());

System.out.println(**"Method arguments : "**

+ Arrays.toString(methodInvocation.getArguments()));

System.out.println(**"AOPAroundMethod : Before method call."**);

**try** {

// proceed to original method call

Object result = methodInvocation.proceed();

// same with AfterReturningAdvice

System.out.println(**"AOPAroundMethod : after call."**);

return result;

} **catch** (IllegalArgumentException e) {

System.out.println(**"AOPAroundMethod : Throw exception call."**);

**throw** e;

}

}

}

Here is the xml Bean configuration file to install the Around Advice AOP.

<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-2.5.xsd">

<bean id=**"myService"** **class**=**"com.java2s.common.PrinterService"**>

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

<property name=**"url"** value=**"http://www.java2s.com"** />

</bean>

<bean id=**"aopAroundMethodBean"** **class**=**"com.java2s.common.AOPAroundMethod"** />

<bean id=**"myServiceProxy"**

**class**=**"org.springframework.aop.framework.ProxyFactoryBean"**>

<property name=**"target"** ref=**"myService"** />

<property name=**"interceptorNames"**>

<list>

<value>aopAroundMethodBean</value>

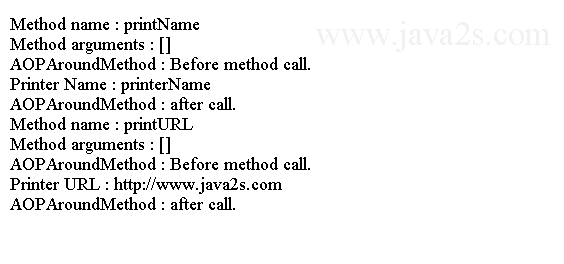
</list>

</property>

</bean>

</beans>

Run the main function, here is the output.

   
  
[Download AOP\_Around.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/AOP_Around.zip)

# Spring Tutorial - Spring Object-XML Mapping

* [« Previous](http://www.java2s.com/Tutorials/Java/Spring/0400__Spring_AOP.htm)

The following code shows how to use Castor to do Java Object to XML Mapping.

## Project Dependency

In order to use castor, add the following dependencies to pom.xml file.

<properties>

<spring.version>3.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<!-- Spring 3 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>

<!-- spring oxm -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-oxm</artifactId>

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

</dependency>

<!-- Uses Castor **for** XML -->

<dependency>

<groupId>org.codehaus.castor</groupId>

<artifactId>castor</artifactId>

<version>1.2</version>

</dependency>

<!-- Castor need this -->

<dependency>

<groupId>xerces</groupId>

<artifactId>xercesImpl</artifactId>

<version>2.8.1</version>

</dependency>

</dependencies>

## Java Bean

Here is a simple Java bean for doing the xml mapping

package com.java2s.common;

**public** **class** Employee {

String name;

int age;

boolean trained;

String address;

**public** String getName() {

return name;

}

**public** void setName(String name) {

this.name = name;

}

**public** int getAge() {

return age;

}

**public** void setAge(int age) {

this.age = age;

}

**public** boolean isTrained() {

return trained;

}

**public** void setTrained(boolean trained) {

this.trained = trained;

}

**public** String getAddress() {

return address;

}

**public** void setAddress(String address) {

this.address = address;

}

}

## Marshaller and Unmarshaller

The following code handles the mapping via Spring's oxm interfaces : Marshaller and Unmarshaller.

package com.java2s.common;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import javax.xml.transform.stream.StreamResult;

import javax.xml.transform.stream.StreamSource;

import org.springframework.oxm.Marshaller;

import org.springframework.oxm.Unmarshaller;

**public** **class** XMLConverter {

**private** Marshaller marshaller;

**private** Unmarshaller unmarshaller;

**public** Marshaller getMarshaller() {

return marshaller;

}

**public** void setMarshaller(Marshaller marshaller) {

this.marshaller = marshaller;

}

**public** Unmarshaller getUnmarshaller() {

return unmarshaller;

}

**public** void setUnmarshaller(Unmarshaller unmarshaller) {

this.unmarshaller = unmarshaller;

}

**public** void convertFromObjectToXML(Object object, String filepath)

throws IOException {

FileOutputStream os = null;

**try** {

os = **new** FileOutputStream(filepath);

getMarshaller().marshal(object, **new** StreamResult(os));

} finally {

**if** (os != null) {

os.close();

}

}

}

**public** Object convertFromXMLToObject(String xmlfile) throws IOException {

FileInputStream is = null;

**try** {

is = **new** FileInputStream(xmlfile);

return getUnmarshaller().unmarshal(**new** StreamSource(is));

} finally {

**if** (is != null) {

is.close();

}

}

}

}

## Spring Configuration

In Spring's bean configuration file, inject CastorMarshaller as the XML binding framework.

<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-3.0.xsd">

<bean id=**"XMLConverter"** **class**=**"com.java2s.common.XMLConverter"**>

<property name=**"marshaller"** ref=**"castorMarshaller"** />

<property name=**"unmarshaller"** ref=**"castorMarshaller"** />

</bean>

<bean id=**"castorMarshaller"** **class**=**"org.springframework.oxm.castor.CastorMarshaller"** />

</beans>

## Example

Here is the code run the application.

package com.java2s.common;

import java.io.IOException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**private** **static** **final** String XML\_FILE\_NAME = **"customer.xml"**;

**public** **static** void main(String[] args) throws IOException {

ApplicationContext appContext = **new** ClassPathXmlApplicationContext(**"SpringBeans.xml"**);

XMLConverter converter = (XMLConverter) appContext.getBean(**"XMLConverter"**);

Employee customer = **new** Employee();

customer.setName(**"java2s"**);

customer.setAge(99);

customer.setTrained(true);

customer.setAddress(**"This is address"**);

//from object to XML file

converter.convertFromObjectToXML(customer, XML\_FILE\_NAME);

//from XML to object

Employee customer2 = (Employee)converter.convertFromXMLToObject(XML\_FILE\_NAME);

System.out.println(customer2);

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/E/EXAMPLE__08F32967D8BACCBE8129.PNG

The following XML file "customer.xml" is generated in the project root folder.

File : customer.xml

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

<customer trained=**"true"** age=**"99"**>

<address>This is address</address>

<name>java2s</name>

</customer>

[Download Castor\_XML\_Mapping.zip](http://www.java2s.com/Tutorials/JavaDownload/Spring/Castor_XML_Mapping.zip)

## Castor XML Mapping

To control which field should use as attribute or element, use Castor XML mapping file to define the relationship between Object and XML.

The following code creates the mapping file, mapping.xml. Put it into our project classpath.

<mapping>

<**class** name=**"com.java2s.core.model.Employee"**>

<map-to xml=**"customer"** />

<field name=**"age"** type=**"integer"**>

<bind-xml name=**"age"** node=**"attribute"** />

</field>

<field name=**"trained"** type=**"boolean"**>

<bind-xml name=**"trained"** node=**"element"** />

</field>

<field name=**"name"** type=**"string"**>

<bind-xml name=**"name"** node=**"element"** />

</field>

<field name=**"address"** type=**"string"**>

<bind-xml name=**"address"** node=**"element"** />

</field>

</**class**>

</mapping>

In Spring bean configuration file, inject above mapping.xml into CastorMarshaller via "mappingLocation".

<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-3.0.xsd">

<bean id=**"XMLConverter"** **class**=**"com.java2s.core.XMLConverter"**>

<property name=**"marshaller"** ref=**"castorMarshaller"** />

<property name=**"unmarshaller"** ref=**"castorMarshaller"** />

</bean>

<bean id=**"castorMarshaller"** **class**=**"org.springframework.oxm.castor.CastorMarshaller"** >

<property name=**"mappingLocation"** value=**"classpath:mapping.xml"** />

</bean>

</beans>