**Core Spring Framework**

**Part 1**

**Participants Notes**

**Why another Framework?**

This is the first question that comes up to everyone’s mind even before showing interest in knowing what is Spring.

There are many Java frameworks available in market; widely accepted ones are struts as MVC framework, Hibernate as ORM framework, EJB and frameworks like velocity and tiles at a view layer. Each of these frameworks serves specific purpose and also requires a container. And integrating them in one application is a pain for programmer. Spring unlike those frameworks is a container as well as a framework and supports integration with all other frameworks by means of some configuration. It is framework because it has its own implementation of MVC and container because it can manage your implementation of ORM and MVC as well as it also integrates well with EJBs and in that case you don't require EJB container. Spring becomes container of EJBs there.

**What is Spring Framework?**

* The Spring Framework is an open source application framework that aims to make J2EE development easier.
* Other frameworks such as Struts or Hibernate are single-tier frameworks.
* Spring aims to help to structure whole applications in a consistent, productive manner, pulling together best-of-breed single-tier frameworks to create a coherent architecture.
* We can use Spring in the stand-alone application development as well as Web development.

Basically, Spring framework was created as an alternative to EJB development. But, later it has been enriched to support both stand-alone as well as web development.

Some of the issues with J2EE applications:

**Some issues with J2EE applications:**

1. **J2EE applications contain lot of plumbing code:**

Many applications contain lot of codes that does not do anything: JNDI lookup code, Transfer Objects, try/catch blocks to acquire and release JDBC resources, EJB life cycle methods such as ejbCreate(), ejbPostCreate(), ejbActivate() etc. .

Writing and maintaining such plumbing code would waste the developers’ time and energy. Rather, if they focus on the application’s business domain logic, their time and energy would be utilized effectively. This is one of the key advantages of Spring framework.

1. **Many J2EE applications use a distributed object model which is not appropriate most of the times:**

This is another area where excessive code and code duplication is found in the application. It's also conceptually wrong in many cases; internally distributed applications are more complex than co-located applications, and often much less performant. If your business requirements need a distributed architecture, you need to implement a distributed architecture and accept the trade-off that incurs. But you shouldn't do so without a compelling reason. Spring helps the developers in such scenarios.

1. **EJB component model is complex:**

Think about writing home, remote interfaces, deployment descriptors, CMP and BMP even for a simple application. It really increases the complexity and on top of this, the developers should be expert in EJB programming and also you need an application server. But, spring is so easy to develop applications with and also no need for application servers. It can work stand-alone as well as with simple web container like tomcat.

**So, what is the alternative?**

Many J2EE APIs and services are cumbersome to use. J2EE does a great job of standardizing low-level infrastructure, solving such problems as how can Java code access transaction management without dealing with the details of XA transactions. But J2EE does not provide an easily usable view for application code.

**That is the role of an application framework, such as Spring.**

**Benefits of Spring framework:**

* Spring offers services for use not only in a single tier, but throughout the application.
* Spring aims to take away much of the pain resulting from the issues in the list we've seen, by simplifying the programming model
* Spring enables you to enjoy the key benefits of J2EE, while minimizing the complexity encountered by application code.
* The essence of Spring is in providing enterprise services to Plain Old Java Objects (POJOs). This is particularly valuable in a J2EE environment, but application code delivered as POJOs is naturally reusable in a variety of runtime environments.
* Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.
* Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
* Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks
* Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
* Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

**Spring – another look:**

**Lightweight :** **Spring** is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.

**Inversion of control (IoC) :** Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.

**Aspect oriented (AOP) :** Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.

**Container :** **Spring** contains and manages the life cycle and configuration of application objects.

**Framework :** **Spring** provides most of the intra functionality leaving rest of the coding to the developer.

**Overview of Spring Framework:**

**Dependency Injection (DI):**

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection helps in gluing these classes together and same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework.

**Aspect Oriented Programming (AOP):**

One of the key components of Spring is the Aspect oriented programming (AOP) framework. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

**What are the different modules in Spring framework?**

There are about 20 modules which can be used based on application requirement.



The Core container module

Application context module

AOP module (Aspect Oriented Programming)

JDBC abstraction and DAO module

O/R mapping integration module (Object/Relational)

Web module

*MVC framework* module

**Core Container Module:**

The Core Container consists of the Core, Beans, Context, and Expression Language modules whose detail is as follows:

* The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
* The Bean module provides BeanFactory which is a sophisticated implementation of the factory pattern.
* The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
* The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime.

This module provides the fundamental functionality of the spring framework. In this module **BeanFactory** is the heart of any spring-based application. The entire framework was built on the top of this module. This module makes the Spring container.

**Data Access/Integration module:**

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows:

* The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.
* The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
* The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The Java Messaging Service JMS module contains features for producing and consuming messages.
* The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

**What is Application context module?**

The Application context module makes spring a framework. This module extends the concept of *BeanFactory*, providing support for internationalization (I18N) messages, application lifecycle events, and validation. This module also supplies many enterprise services such JNDI access, *EJB integration*, remoting, and scheduling. It also provides support to other framework.

**About AOP Module:**

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between Spring and other AOP frameworks. This module also introduces metadata programming to Spring. Using Spring’s metadata support, we will be able to add annotations to our source code that instruct Spring on where and how to apply aspects.

**About web module:**

The Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules whose detail is as follows:

* The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
* The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications.
* The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application.
* The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

This module is built on the application context module, providing a context that is appropriate for web-based applications. This module also contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Jakarta Struts.

Spring comes with a full-featured MVC framework for building web applications. Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring’s MVC framework uses IoC to provide for a clean separation of controller logic from business objects. It also allows you to decoratively bind request parameters to your business objects. It also can take advantage of any of Spring’s other services, such as I18N messaging and validation.

**Other Modules:**

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules whose detail is as follows:

* The AOP module provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The Aspects module provides integration with AspectJ which is again a powerful and mature aspect oriented programming (AOP) framework.
* The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.
* The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

**How to setup the Spring environment?**

**Step 1: Install JDK**

**Step 2: Install Apache Common Logging API**

Download and install from <http://commons.apache.org/logging/>

After installing into your convenient location, it will look something like this:



**Step 3: Setup Eclipse IDE:**

If you don’t have the eclipse setup, download and install from <http://www.eclipse.org/downloads/>.

**Step 4: Setup Spring Framework libraries:**

Download Spring from [www.springsource.org/download](http://www.springsource.org/download).

After installation into your convenient location, it will look something like this:



All Spring libraries are available under dist directory. So, make all these libraries available for your project by including in the classpath.

**First Spring Example:**

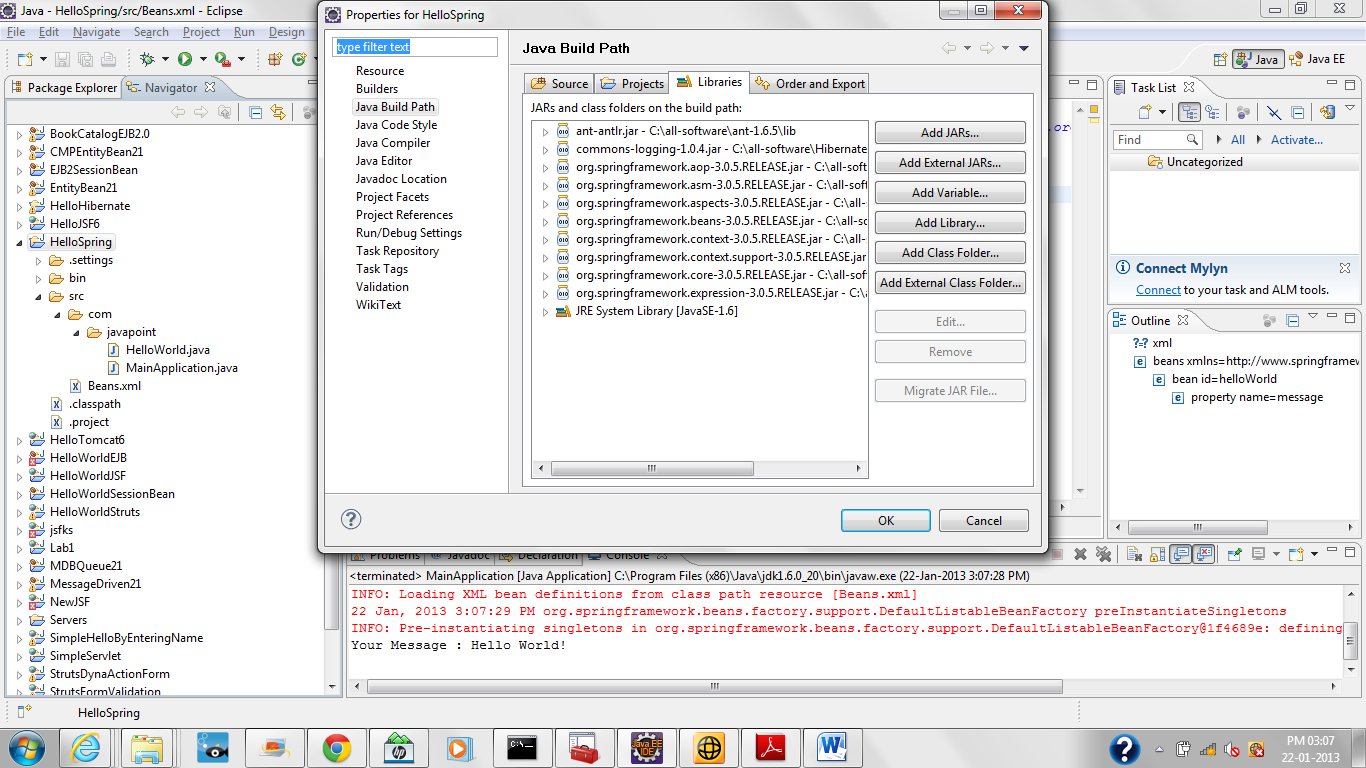
**Step1:**

Create Java project from Eclipse

**Step 2:**

Add required libraries

Add the following jar files from Eclipse. It should look like this:



**Step 3:**

Create required Java source files:

**Program 1: HelloWorld.java**

**package** com.oracle;

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

**private** String message;

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

**this**.message = message;

}

**public** **void** getMessage(){

System.*out*.println("Your Message : " + message);

}

}

**Program 2: MainApplication.java**

**package** com.oracle;

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

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

**public** **class** MainApplication {

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

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

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

}

There are following two important points to note about the main program:

1. First step is to create application context where we used framework API ClassPathXmlApplicationContext(). This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects ie. beans mentioned in the configuration file.

2. Second step is used to get required bean using getBean() method of the created context. This method uses bean ID to return a generic object which finally can be casted to actual object. Once you have object, you can use this object to call any class method.

**Step 4: Create Bean configuration file.**

Create a file Beans.xml in the “src” directory of your application.

**Beans.xml:**

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

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

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

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

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

<bean id=*"helloWorld"* class=*"com.oracle.HelloWorld"*>

<property name=*"message"* value=*"Welcome to Spring!"*/>

</bean>

</beans>

**Step 5:**

Run the program by right clicking the MainApplication.java Run As … Java Application. If everything is correct, you would see the following output.

Your Message : Welcome to Spring!

Fantastic! You have created your first Spring Application successfully. You can see the flexibility of above Spring application by changing the value of "message" property and keeping both the source files unchanged.

**Spring IoC Containers:**

When we introduced Spring, we said that Spring acts like a container too. The Spring container is at the core of the Spring framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation to destruction. The Spring container uses dependency injection (DI) technique to manage the components that make up an application.

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



Spring has the following two types of containers.

1. **Spring BeanFactory Container:**

This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

1. **Spring ApplicationContext Container:**

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContextinterface.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

**More details about these containers:**

1. **Spring BeanFactory Container:**

As already said, this is the simplest container. There are number of implementations that come along with Spring. The most widely used implementation is XmlBeanFactory class. This container reads the metadata from the xml file and uses to create a fully functional application. The BeanFactory container is mostly preferred when the application is small or when the resources are limited or desktop stand-alone applications.

**An example program using BeanFactory container:**

**Step 1:**

Same as above i.e. create Java project from Eclipse

**Step 2:**

Same as above. Add the required libraries

**Step 3:**

Create required Java source files:

**Program 1: HelloWorldBeanFactory.java**

Same as above HelloWorld.java except the source code name (it is HelloWorldBeanFactory.java)

**Program 2: MainApplication.java**

package com.oracle;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

public class MainApplication {

public static void main(String[] args) {

XmlBeanFactory factory = new XmlBeanFactory

(new ClassPathResource("Beans.xml"));

HelloWorldBeanFactory obj = (HelloWorldBeanFactory) factory.getBean("helloWorld");

obj.getMessage();

}

}

**The following points to be noted about this MainApplication.java program.**

1. First step is to create factory object where we used framework API XmlBeanFactory() to create the factory bean and ClassPathResource() API to load the bean configuration file available in CLASSPATH. The XmlBeanFactory() API takes care of creating and initializing all the objects ie. beans mentioned in the configuration file.
2. Second step is used to get required bean using getBean() method of the created bean factory object. This method uses bean ID to return a generic object which finally can be casted to actual object. Once you have object, you can use this object to call any class method.

**Step 4: Create Bean configuration file (Beans.xml)**

Place Beans.xml under the “src” directory of your application.

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

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

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

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

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

<bean id="helloWorld" class="com.oracle.HelloWorldBeanFactory">

<property name="message" value="Welcome to Spring. Program using BeanFactory!"/>

</bean>

</beans>

**Step 5:**

Run the program by right clicking the MainApplication.java Run As … Java Application. If everything is correct, you would see the following output.

Your Message: Welcome to Spring. Program using BeanFactory!

1. **Spring ApplicationContext Container:**

This is Spring’s more advanced container. It works similar to BeanFactory in the sense that it can load bean definitions, wire beans together and dispense beans upon request. But, it has enterprise-specific functionalities such as resolving textual messages from the properties files, publishing application events to interested event listeners. This container is defined by org.springframework.context.ApplicationContext interface.

Since ApplicationContext container has all the functionalities of BeanFactory plus additional enterprise-specific functionalities, it is preferred over BeanFactory. But still, BeanFactory can be used in case of light weight applications.

The following are the three most commonly used implementations available for ApplicationContext container.

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

We have already seen an example using ClassPathXmlApplicationContext. Let’s discuss about XmlWebApplicationContext later. Now, let’s see another example using FileSystemXmlApplicationContext.

**An example program using FileSystemXmlApplicationContext implementation:**

**Step 1:**

Same as above i.e. Create a Java project in Eclipse.

**Step 2:**

Same as above. Add the required libraries.

**Step 3:**

Create required Java source files:

**Program 1: HelloSpringFileSystem.java**

Same as above HelloWorld.java except that the source code name (it is HelloSpringFileSystem.java)

**Program 2: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplicationContext("C:/guru/FileSystemXmlSpring/src/Beans.xml");

HelloSpringFileSystem obj = (HelloSpringFileSystem) context.getBean("helloWorld");

obj.getMessage();

}

}

**The points to be noted here are:**

1. First step is to create context object with FileSystemXmlApplicationContext api that refers to the bean definition (Beans.xml), where the bean is loaded from the given absolute path (c:/guru/Project-name/src/Beans.xml).
2. Second step is using the context object, get the bean using the bean id (helloWorld), it will return the generic object. Then, we do the casting on this generic object to HelloSpringFileSystem. Then, call the method getMessage() on this object.

**Step 4: Create the Bean configuration file (Beans.xml) – under “src” directory.**

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

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

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

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

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

<bean id="helloWorld" class="com.oracle.HelloSpringFileSystem">

<property name="message" value="Hello World!"/>

</bean>

</beans>

**Step 5:**

Run the program by right clicking the MainApplication.java Run As … Java Application. If everything is correct, you would see the following output.

Your Message : From HelloSpringFileSystem Hello World!

**Spring Bean Definition:**

Spring beans are the objects that form the backbone of the application and are managed by the IoC containers. These beans are created from the configuration metadata that you supply to the container, in the form of xml file. So, the IoC container instantiates, assemble and manage these beans.

Just look at the following bean definition.

<bean id=*"helloWorld"* class=*"com.oracle.HelloSpringFileSystem"*>

<property name=*"message"* value=*"Hello World!"*/>

</bean>

This bean definition contains information called as **configuration metadata** which is required for the IoC container to know:

* How to create a bean
* Bean’s life cycle details
* Bean’s dependencies.

The following are the properties for the beans.

|  |  |
| --- | --- |
| Properties | Descriptions |
| Class | Specifies the bean class to be used to create the bean |
| Name | Specifies the bean id(endifier) |
| Scope | Specifies the scope of the objects created from a bean definition file |
| constructor-arg | This is used for DI, will be discussed later. |
| Properties | This is also used for DI, will be discussed later. |
| autowiring mode | This is also used for DI, will be discussed later. |
| lazy-initialization mode | Informs the container to create the bean instance when it is first requested, rather than at startup |
| initialization method | A callback, will be discussed later. |
| destruction method | A calback to be called when the container containing the bean is destroyed, will be discussed later. |

**Spring Configuration metadata:**

There are three methods to provide configuration metadata to the container.

1. XML based configuration
2. Annotation based configuration
3. Java based configuration

**XML based configuration:**

We have already seen the XML based configuration with various examples. All bean definitions we have provided so far (Beans.xml) are examples of XML based configuration.

Let’s see about annotation based configuration and Java based configuration later, because before seeing these, we need to understand some other stuff.

**Spring Bean Scopes:**

When defining a bean in Beans.xml, there is an option of declaring a scope for that bean. For example, to request Spring to create a new bean instance everytime it is needed, we can declare the bean’s scope as **“prototype”**. Similarly, to request Spring to return the same bean instance evertime it is needed, we can declare the bean’s scope as **“singleton”.**

There are five bean scopes available:

|  |  |
| --- | --- |
| **Scope** | **Descriptions** |
| Singleton | Default scope.  This creates a single instance of a bean and returns the same instance per Ioc Container. |
| Prototype | This creates a new instance of a bean everytime it is needed. It creates any number of bean instances. |
| Request | This is possible with web application. This scope is associated with HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This is possible with web application. This scope is associated with HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This is possible with web application. This scope is associated with global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

**Singleton scope - Example**

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

<!-- other configurations go here -->

</bean>

**Step 1:**

Create a Java project in Eclipse

**Step 2:**

Add the required libraries

**Step 3:**

Create the java source files.

**Program 1:** Same as above HelloWorld.java with setMessage(..) and getMessage(..) methods

**Program 2:** MainApplication.java

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj1 = (HelloWorld) context.getBean("helloWorld");

Obj1.setMessage("I am the object 1");

Obj1.getMessage();

HelloWorld obj2 = (HelloWorld) context.getBean("helloWorld");

Obj2.getMessage();

}

}

**Step 4:**

Create bean definition file (Beans.xml) in src.

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

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

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

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

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

<bean id="helloWorld" class="com.oracle.HelloWorld"

scope="**singleton**">

</bean>

</beans>

**Step 5:**

Run the MainApplication.java, you would see the following output

Your Message : I am the object 1

Your Message : I am the object 1

**Try yourself…prototype scope**

Do the same program above, just change scope from “singleton” to “prototype” in Beans.xml and verify that you get the following output.

Your Message : I am the object 1

Your Message : null

**Spring Bean Life Cycle:**

The life cycle of a Spring bean is easy to understand.

When a bean is instantiated –

it may be required to perform some initialization

When the bean is no longer required and is removed from the container –

some cleanup may be required.

We are going to discuss two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.

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

The org.springframework.beans.factory.InitializingBean interface specifies a single method:

void afterPropertiesSet() throws Exception;

So, you can implement the above interface and all the required initialization code can be inserted inside the afterPropertiesSet method.

public class MyBean implements IntializingBean {

public void afterPropertiesSet() throws Exception {

// initialization code goes here

}

}

In case of XML-based configuration metadata, you can use init-method attribute to specify the name of the method to be called. For example:

<bean id="myBean"

class="com.oracle.MyBean" init-method="init"/>

Following the java class:

public class MyBean

{

public void init()

{

// initialization code

}

}

Similarly, we have the destruction callbacks.

The org.springframework.beans.factory.DisposableBean interface specifies a single method:

void destroy() throws Exception;

The class definition can be:

public class MyBean implements DisposableBean

{

public void destroy() throws Exception

{

// destruction code goes here

}

}

In case of XML-based configuration metadata, we have the destroy-method attribute to specify which method to be called upon destructing the bean.

For example:

<bean id=”myBean"

class="com.oracle.MyBean" destroy-method="destroy"/>

The class definition can be:

public class MyBean

{

public void destroy()

{

//destruction code goes here

}

}

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

**Example:**

Now, let’s quickly do a simple program to demonstrate the above. I am not going to repeat the steps. Just see below the required java class and XML bean definition files.

**Program 1: HelloWorld.java**

package com.oracle;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

public void init(){

System.out.println("Bean init method is getting called");

}

public void destroy(){

System.out.println("Bean destroy method is getting calleed");

}

}

**Program 2: MainApplication.java**

package com.oracle;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

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

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

**Beans.xml:**

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

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

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

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

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

<bean id="helloWorld" class="com.oracle.HelloWorld" init-method="init"

destroy-method="destroy">

<property name="message" value="Hello World!"/>

</bean>

</beans>

**Note:**

If you have too many beans having initialization and/or destroy methods with the same names, no need to declare separate init-method and destroy-method on each beans. Rather, you can use default-init-method and default-destroy-method attributes.

**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-3.0.xsd"

default-init-method="init"

default-destroy-method="destroy">

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

//other configurations here….

</bean>

</beans>

**Spring Bean Definition Inheritance:**

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

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

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

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

**Example:**

**Beans.xml:**

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

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

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

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

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

<bean id="helloWorld" class="com.oracle.HelloWorld">

<property name="message1" value="Hello World One!"/>

<property name="message2" value="Hello World Two!"/>

</bean>

<bean id="helloIndia" class="com.oracle.HelloIndia"

parent="helloWorld">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Welcome India!"/>

</bean>

</beans>

**HelloWorld.java**

package com.oracle;

public class HelloWorld {

private String message1;

private String message2;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void getMessage1(){

System.out.println("World Message1 : " + message1);

}

public void getMessage2(){

System.out.println("World Message2 : " + message2);

}

}

**HelloIndia.java**

package com.oracle;

public class HelloIndia {

private String message1;

private String message2;

private String message3;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void setMessage3(String message){

this.message3 = message;

}

public void getMessage1(){

System.out.println("India Message1 : " + message1);

}

public void getMessage2(){

System.out.println("India Message2 : " + message2);

}

public void getMessage3(){

System.out.println("India Message3 : " + message3);

}

}

**MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.getMessage1();

objA.getMessage2();

HelloIndia objB = (HelloIndia) context.getBean("helloIndia");

objB.getMessage1(); // child’s property

objB.getMessage2(); // parent’s property

objB.getMessage3(); // child’s property

}

}

**The output would be:**

World Message1 : Hello World One!

World Message2 : Hello World Two!

India Message1 : Hello India!

India Message2 : Hello World Two!

India Message3 : Welcome India!

If you observed here, we did not pass message2 while creating "helloIndia" bean, but it got passed because of Bean Definition Inheritance

**Bean Definition template:**

Recall the abstract class and child class which implements the abstract class concept from core java. The same can be achieved in Spring using bean definition template.

You can create a Bean definition template which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should not specify class attribute and should specify abstract attribute with a value of true as shown below:

**Beans.xml:**

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

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

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

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

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

<bean id="beanTeamplate" abstract="true">

<property name="message1" value="Hello World One!"/>

<property name="message2" value="Hello World Two!"/>

<property name="message3" value="Welcome India!"/>

</bean>

<bean id="helloIndia" class="com.oracle.HelloIndia"

parent="beanTeamplate">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Welcome India!"/>

</bean>

</beans>

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as abstract. When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions.

**Spring Dependency Injection (DI)**

Every java based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.

Consider you have an application which has a ClassRoom component and you want to provide Student. Your standard code would look something like this:

public class ClassRoom {

private Student student;

public ClassRoom() {

student = new Student();

}

}

Here, there is a dependency between ClassRoom and Student class.

This is the standard way. Here, the ClassRoom class has taken the responsibility of creating the Student object. But ClassRoom class may not know the implementation of Student class. In such cases, we will follow the concept called IoC (Inversion of Control), where creation of Student object is delegated outside ClassRoom class. Now, it would look like this:

public class ClassRoom {

private Student student;

public ClassRoom(Student student) {

this.student = student;

}

}

Now, the class ClassRoom need not worry about the creation of Student object. The Student will be implemented independently and will be provided to ClassRoom at the time of ClassRoom instantiation and the entire process is controlled by Spring framework.

Here, we have removed the total control from ClassRoom and kept it somewhere else (ie. XML configuration file) and the dependency ( ie. class Student) is being injected into the class ClassRoom through a Class Constructor. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through Setter Methods of ClassRoom class where we will create Student instance and this instance will be used to call setter methods to initialize ClassRoom’s properties.

There are two types of DI:

1. **Constructor based dependency injection**

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

1. **Setter based dependency injection**

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

**Note:** You can use any DI, but use constructor arguments for mandatory dependencies and setters for optional dependencies.

1. **Constructor based dependency injection**

**Program 1: ClassRoom.java**

package com.oracle;

public class ClassRoom {

private Student student;

public ClassRoom(Student student) {

System.out.println("Inside ClassRoom constructor." );

this.student = student;

}

public void callStudent() {

student.callingStudent();

}

}

**Program 2: Student.java**

package com.oracle;

public class Student {

public Student(){

System.out.println("Inside Student constructor." );

}

public void callingStudent() {

System.out.println("Inside callingStudent." );

}

}

**Program 3: MainApplication.java:**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

ClassRoom cr = (ClassRoom) context.getBean("classRoom");

cr.callStudent();

}

}

**Beans.xml:**

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

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

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

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

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

<!-- Definition for ClassRoom bean -->

<bean id="classRoom" class="com.oracle.ClassRoom">

<constructor-arg ref="student"/>

</bean>

<!-- Definition for Student bean -->

<bean id="student" class="com.oracle.Student">

</bean>

</beans>

**Constructor argument ambiguities:**

If we pass more than one arguments to the constructor, then in which order we should pass those arguments? To resolve this ambiguity, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor.

**Example:**

package a.b

public class Foo {

public Foo(Material mat, Metal met) {

// ...

}

}

In Bean definition, we should give like this:

<beans>

<bean id="foo" class="a.b.Foo">

<constructor-arg ref="mat"/>

<constructor-arg ref="met"/>

</bean>

<bean id="mat" class="a.b.Material"/>

<bean id="met" class="a.b.Metal"/>

</beans>

**Another example:**

If the constructor accepts the different data types, then how to pass those?

package a.b;

public class Foo {

public Foo(String name, int age) {

// ...

}

}

The bean definition can be:

<beans>

<bean id="myBean" class="com.oracle.MyBean">

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

<constructor-arg type="int" value="35"/>

</bean>

</beans>

Another best way is using the index attribute to specify exactly the index of the constructor arguments. The index is 0 based.

<beans>

<bean id="myBean" class="com.oracle.MyBean">

<constructor-arg index="0" value="Guru"/>

<constructor-arg index="1" value="35"/>

</bean>

</beans>

**Note**: If you pass the reference to an object, use **ref** attribute of <constructor-arg> tag and if you pass the value directly then use **value** attribute of <constructor-arg> tag.

1. **Setter based dependency injection:**

**Program 1: ClassRoom.java**

package com.oracle;

public class ClassRoom {

private Student student;

// a setter method to inject the dependency.

public void setStudent(Student student) {

System.out.println("Inside setStudent." );

this.student = student;

}

// a getter method to return student

public Student getStudent() {

return student;

}

public void callStudent() {

student.callingStudent();

}

}

**Program 2: Student.java**

package com.oracle;

public class Student {

public Student(){

System.out.println("Inside Student constructor." );

}

public void callingStudent() {

System.out.println("Inside callingStudent." );

}

}

**Program 3: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =new ClassPathXmlApplicationContext("Beans.xml");

ClassRoom cr = (ClassRoom) context.getBean("classRoom");

cr.callStudent();

}

}

**Beans.xml:**

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

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

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

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

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

<!-- Definition for classRoom bean -->

<bean id="classRoom" class="com.oracle.ClassRoom">

<property name="student" ref="student"/>

</bean>

<!-- Definition for student bean -->

<bean id="student" class="com.oracle.Student">

</bean>

</beans>

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

**XML Configuration using p-namespace:**

If you have many setter methods then it is convenient to use p-namespace in the Beans.xml file.

**Example:**

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

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

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

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

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

<bean id="employee" class="com.oracle.Employee">

<property name="name" value="Gurumurthy"/>

<property name="address" ref="address"/>

</bean>

<bean name="address" class="com.oracle.Address">

<property name="name" value="Bangalore"/>

</bean>

</beans>

This can be written as below:

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

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

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

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

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

<bean id="employee" class="com.oracle.Employee"

p:name="Gurumurthy"

p:address-ref="address"/>

</bean>

<bean name="address" class="com.oracle.Address"

p:name="Bangalore"/>

</bean>

</beans>

**Spring injecting Collections:**

You can use Java collections to be injected into beans. Spring offers four types of collection configuration elements which are as follows:

|  |  |
| --- | --- |
| Elements | Descriptions |
| <list> | This helps in wiring ie injecting a list of values, allowing duplicates. |
| <set> | This helps in wiring a set of values but without any duplicates. |
| <map> | This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| <props> | This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

**Example:**

**Program 1: CollectionExample.java**

package com.oracle;

import java.util.\*;

public class CollectionExample {

List cityList;

Set citySet;

Map cityMap;

Properties cityProp;

// a setter method to set List

public void setCityList(List cityList) {

this.cityList = cityList;

}

// prints and returns all the elements of the list.

public List getCityList() {

System.out.println("List Elements :" + cityList);

return cityList;

}

// a setter method to set Set

public void setCitySet(Set citySet) {

this.citySet = citySet;

}

// prints and returns all the elements of the Set.

public Set getCitySet() {

System.out.println("Set Elements :" + citySet);

return citySet;

}

// a setter method to set Map

public void setCityMap(Map cityMap) {

this.cityMap = cityMap;

}

// prints and returns all the elements of the Map.

public Map getCityMap() {

System.out.println("Map Elements :" + cityMap);

return cityMap;

}

// a setter method to set Property

public void setCityProp(Properties cityProp) {

this.cityProp = cityProp;

}

// prints and returns all the elements of the Property.

public Properties getCityProp() {

System.out.println("Property Elements :" + cityProp);

return cityProp;

}

}

**Program 2: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

CollectionExample ce=(CollectionExample)context.getBean("collectionExample");

ce.getCityList();

ce.getCitySet();

ce.getCityMap();

ce.getCityProp();

}

}

**Beans.xml:**

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

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

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

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

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

<!-- Definition for CollectionExample -->

<bean id="collectionExample" class="com.oracle.CollectionExample">

<!-- results in a setCityList(java.util.List) call -->

<property name="cityList">

<list>

<value>Bangalore</value>

<value>Hyderabad</value>

<value>Chennai</value>

<value>Chennai</value>

</list>

</property>

<!-- results in a setCitySet(java.util.Set) call -->

<property name="citySet">

<set>

<value>Bangalore</value>

<value>Hyderabad</value>

<value>Chennai</value>

<value>Chennai</value>

</set>

</property>

<!-- results in a setCityMap(java.util.Map) call -->

<property name="cityMap">

<map>

<entry key="1" value="Bangalore"/>

<entry key="2" value="Hyderabad"/>

<entry key="3" value="Chennai"/>

<entry key="4" value="Chennai"/>

</map>

</property>

<!-- results in a setCityProp(java.util.Properties) call -->

<property name="cityProp">

<props>

<prop key="one">Bangalore</prop>

<prop key="two">Hyderabad</prop>

<prop key="three">Chennai</prop>

<prop key="four">Chennai</prop>

</props>

</property>

</bean>

</beans>

**You should get the following output:**

List Elements :[Bangalore, Hyderabad, Chennai, Chennai]

Set Elements :[Bangalore, Hyderabad, Chennai]

Map Elements :{1=Bangalore, 2=Hyderabad, 3=Chennai, 4=Chennai}

Property Elements :{two=Hyderabad, one=Bangalore, three=Chennai, four=Chennai}

**Injecting null and empty string values**

If you need to pass an empty string as a value then you can pass it as follows:

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

<property name="email" value=""/>

</bean>

This is equivalent to the Java code:

exampleBean.setEmail("").

If you need to pass an NULL value then you can pass it as follows:

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

<property name="email"><null/></property>

</bean>

This is equivalent to the Java code:

exampleBean.setEmail(null);

**Spring Beans auto-wiring:**

We have seen that how to declare beans using the <bean> element and inject <bean> with using <constructor-arg> and <property> elements in XML configuration file. The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

**Auto-wiring modes:**

|  |  |
| --- | --- |
| **Modes** | **Descriptions** |
| No | This is the default mode, which means no auto-wiring. You should use explicit bean reference for wiring. There is nothing to be done for this mode, whatever we have seen in DI session, is what this is. |
| Byname | Auto-wiring by property name.  Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.  “Autowiring by Name” means, if the name of a bean is same as the name of other bean property, auto wire it.  For example, if a “customer” bean exposes an “address” property, Spring will find the “address” bean in current container and wire it automatically. And if no matching found, just do nothing. |
| byType | Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown. |
| Constructor | Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| Autodetect | Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType. |

**Note:**

You can use byType or constructor autowiring mode to wire arrays and other typed-collections.

**Problems with auto-wiring:**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

**Other limitations of auto-wiring:**

1. **Overriding possibility:** If you specify dependencies using <constructor-arg> and <property> settings, it will override auto-wiring.
2. **Primitive data types:** You cannot use auto-wiring for primitive data types, Strings and classes.
3. **Confusing nature:**  Autowiring may not be perfect always, so prefer explicit wiring.

**Example of autowiring (byName):**

**Program 1: Customer.java**

package com.oracle;

public class Customer {

private Address address;

private String name;

public void setAddress( Address address ){

this.address = address;

}

public Address getAddress() {

return address;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void displayAddress() {

address.displayingAddress();

}

}

**Program 2: Address.java**

package com.oracle;

public class Address {

public Address() {

System.out.println("Inside Address constructor." );

}

public void displayingAddress() {

System.out.println("Inside displayingAddress." );

}

}

**Program 3: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

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

ct.displayAddress();

}

}

**Beans.xml:**

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

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

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

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

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

<!-- Definition for Customer bean -->

<bean id="customer" class="com.oracle.Customer">

<property name="address" ref="address" />

<property name="name" value="Gurumurthy" />

</bean>

<!-- Definition for address bean -->

<bean id="address" class="com.oracle.Address">

</bean>

</beans>

If you use auto-wiring byName, the above Beans.xml would be like this:

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

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

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

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

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

<!-- Definition for Customer bean -->

<bean id="customer" class="com.oracle.Customer" autowire="byName">

<property name="name" value="Gurumurthy" />

</bean>

<!-- Definition for address bean -->

<bean id="address" class="com.oracle.Address">

</bean>

</beans>

In this case, the Spring container will automatically locate the address bean and auto-wire with the Customer bean. Because, the Customer bean has the property Address and there is a corresponding bean by name Address, so it will auto-wire.

**Example of autowiring (byType):**

It is very similar to byName. The only difference is instead of going by the name, it will go by the type of the attribute that the bean has.

**Program 1: Person.java**

package com.oracle;

public class Person

{

private Ability ability;

//...

}

**Program 2: Ability.java**

package com.oracle;

public class Ability

{

private String skill;

//...

}

**Note:**

If you see the Person.java, it has one of the attributes as type of Ability class.

**Beans.xml:**

<bean id="person" class="com.oracle.Person" autowire="byType" />

<bean id="sing" class="com.oracle.Ability" >

<property name="skill" value="Sing" />

</bean>

**Important:**

Wait, what if you have two beans with same data type of class “ability”?

It will throw the following exception

Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException:

**In autowiring by type mode, you just have to make sure only one unique data type of bean is declared.**

**Example of autowiring (by constructor):**

This mode is very similar to byType, but it applies to constructor arguments. Spring container looks at the beans on which autowire attribute is set to constructor in the XML configuration file. It then tries to match and wire its constructor's argument with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

**Program 1: Customer.java**

package com.oracle;

public class Customer {

private Address address;

private String name;

Customer(Address address, String name)

{

// Constructor with address ref and name as String type.

}

public void setAddress( Address address ){

this.address = address;

}

public Address getAddress() {

return address;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void displayAddress() {

address.displayingAddress();

}

}

**Program 2: Address.java**

package com.oracle;

public class Address {

public Address() {

System.out.println("Inside Address constructor." );

}

public void displayingAddress() {

System.out.println("Inside displayingAddress." );

}

}

**Program 3: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

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

ct.displayAddress();

}

}

**Beans.xml (normal condition):**

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

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

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

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

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

<!-- Definition for customer bean -->

<bean id="customer" class="com.oracle.Customer">

<constructor-arg ref="address" />

<constructor-arg value="Gurumurthy"/>

</bean>

<!-- Definition for address bean -->

<bean id="address" class="com.oracle.Address">

</bean>

</beans>

**Beans.xml (auto-wiring by constructor):**

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

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

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

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

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

<!-- Definition for customer bean -->

<bean id="customer" class="com.oracle.Customer"

autowire="constructor">

<constructor-arg value="Gurumurthy"/>

</bean>

<!-- Definition for address bean -->

<bean id="address" class="com.oracle.Address ">

</bean>

</beans>

**Spring Annotation Based Configuration:**

Starting from Spring 2.5, it is possible to configure the dependency injection using annotations.

What are annotations?

Annotations are basically passing some information about the program to JVM during run-time. These are prefixed with @ symbol. This passed information is called as “meta-data”.

In Spring, Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.

How to turn on annotation wiring?

Just include the following line in Beans.xml

<context:annotation-config/>

Before starting the examples using annotation, let’s see some important annotations.

@Required – This applies to bean property setter methods.

@Autowired – This applies to bean property setter, non-setter methods, constructors and properties.

@Qualifier – This along with @Autowired can be specify exactly which bean needs to be wired.

JSR-250 Annotations – This includes @Resource, @PostConstruct, @PreDestroy annotations.

**Spring @Required Annotation:**

The @Required annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time otherwise the container throws a BeanInitializationException exception. Below is an example to show the use of @Required annotation

**Example:**

**Program 1: Student.java**

package com.oracle;

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

public class Student {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**Program 2: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

Student student = (Student) context.getBean("student");

System.out.println("Name : " + student.getName() );

System.out.println("Age : " + student.getAge() );

}

}

**Beans.xml:**

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

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

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

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

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:annotation-config/>

<!-- Definition for student bean -->

<bean id="student" class="com.oracle.Student">

<property name="name" value="Gurumurthy" />

<!-- try without passing age and check the result -->

<!-- property name="age" value="35"-->

</bean>

</beans>

**Note:**

If you run the above program, it will throw the exception BeanInitializationException and print the following message.

Property ‘age’ is required for bean ‘student’

Now, remove the comment from Beans.xml file for age, run the program, it will print the following output.

Name: Gurumurthy

Age: 35

**Spring @Autowired Annotation:**

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

**@Autowired on Setter Methods**

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

**Example:**

**Program 1: ClassRoom.java**

package com.oracle;

public class ClassRoom {

private Student student;

@Autowired

public void setStudent(Student student) {

System.out.println("Inside setStudent." );

this.student = student;

}

// a getter method to return student

public Student getStudent() {

return student;

}

public void callStudent() {

student.callingStudent();

}

}

**Program 2: Student.java**

package com.oracle;

public class Student {

public Student(){

System.out.println("Inside Student constructor." );

}

public void callingStudent() {

System.out.println("Inside callingStudent." );

}

}

**Program 3: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =new ClassPathXmlApplicationContext("Beans.xml");

ClassRoom cr = (ClassRoom) context.getBean("classRoom");

cr.callStudent();

}

}

**Beans.xml**

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

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

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

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

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

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

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

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

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

<bean id="classRoom" class="com.oracle.ClassRoom" >

</bean>

<bean id="student" class="com.oracle.Student">

</bean>

</beans>

**@Autowired on properties:**

**Example:**

All the above programs are same except the following change in ClassRoom.java. Note that the @Autowired annotation is moved to the property student as below. Rest are all same.

@Autowired

private Student student;

**@Autowired on Constructors:**

You can apply @Autowired annotation on constructors also. This indicates that the constructors should be autowired when creating the bean, even if no <constructor-arg> elements are specified in the bean configuration XML file.

**Example:**

**Program 1: ClassRoom.java**

package com.oracle;

public class ClassRoom {

private Student student;

@Autowired

public ClassRoom(Student student) {

System.out.println("Inside ClassRoom constructor" );

this.student = student;

}

public void callStudent() {

student.callingStudent();

}

}

**Program 2: Student.java**

Same as above

**Beans.xml:**

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

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

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

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

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:annotation-config/>

<bean id="classRoom" class="com.oracle.ClassRoom">

</bean>

<bean id="student" class="com.oracle.Student">

</bean>

</beans>

**@Autowired with (required=false) option:**

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, but, you can turn off the default behavior by using (required=false) option with @Autowired.

**Example:**

**Student.java:**

package com.oracle;

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

public class Student {

private Integer age;

private String name;

@Autowired(required=false)

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Autowired

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**Spring @Qualified Annotation:**

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

**Example:**

**Student.java**

package com.oracle;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**Qualification.java:**

package com.oracle;

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

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

public class Qualification {

@Autowired

@Qualifier("student1")

private Student student;

public Qualification(){

System.out.println("Inside Qualification constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

**MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

Qualification quali = (Qualification) context.getBean("qualification");

quali.printAge();

quali.printName();

}

}

**Beans.xml:**

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

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

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

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

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:annotation-config/>

<bean id="qualification" class="com.oracle.Qualification">

</bean>

<!-- Definition for student1 bean -->

<bean id="student1" class="com.oracle.Student">

<property name="name" value="Gurumurthy" />

<property name="age" value="35"/>

</bean>

<!-- Definition for student2 bean -->

<bean id="student2" class="com.oracle.Student">

<property name="name" value="Raja" />

<property name="age" value="23"/>

</bean>

</beans>

Eventhough, we have defined two beans in the xml file, only “student1” will be autowired.

**Spring JSR-250 Annotations:**

This includes @PostConstruct, @PreDestroy and @Resource annotations.

**@PostConstruct and @PreDestroy annotations:**

We have already seen in the XML based configuration that init-method and destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

Instead of specifying in XML file, you can specify these call back methods in the java file.

**Example:**

**HelloWorld.java**

package com.oracle;

import javax.annotation.\*;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public String getMessage(){

System.out.println("Your Message : " + message);

return message;

}

@PostConstruct

public void init(){

System.out.println("Bean init metod.");

}

@PreDestroy

public void destroy(){

System.out.println("Bean destroy method.”);

}

}

**MainApplication.java**

package com.oracle;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApplication {

public static void main(String[] args) {

AbstractApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

}

**Beans.xml:**

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

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

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

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

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:annotation-config/>

<bean id="helloWorld"

class="com.oracle.HelloWorld"

init-method="init" destroy-method="destroy">

<property name="message" value="Hello World!"/>

</bean>

</beans>

**@Resource annotation:**

You can use @Resource annotation on fields or setter methods.

package com.oracle;

public class ClassRoom {

private Student student;

@Resource(name=”student”)

public void setStudent(Student student) {

System.out.println("Inside setStudent." );

this.student = student;

}

// a getter method to return student

public Student getStudent() {

return student;

}

public void callStudent() {

student.callingStudent();

}

}

**Note:**

If no 'name' is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.

**Spring Java based configuration:**

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

**Example:**

**Program1: ConfigHelloWorld.java**

package com.oracle.

import org.springframework.context.annotation.\*;

@Configuration

public class ConfigHelloWorld {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

This is equivalent to the following xml configuration:

<beans>

<bean id="helloWorld" class="com.oracle.HelloWorld" />

</beans>

**Program2: HelloWorld.java**

package com.oracle;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

**Program3: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(ConfigHelloWorld.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

}

**Note:**

Since, we are using Java-based annotations, we need two more additional jar files to be included in the project classpath.

1. *CGLIB.jar* from the Java installation directory
2. *ASM.jar* library which can be downloaded from *asm.ow2.org*.

After including these jar files, run the MainApplication from Eclipse. If everything is fine, it will display Hello World!

Interestingly, this program does not have any bean configuration xml file.

**Injecting Bean dependencies using Java annotations:**

When the beans have dependencies on one another, specifying such dependencies using Java annotations is very easy. It is as simple as that one bean method calling another.

**Example:**

**Program1: ClassRoomConfig.java**

**package** com.oracle;

**import** org.springframework.context.annotation.\*;

@Configuration

**public** **class** ClassRoomConfig {

@Bean

**public** ClassRoom classRoom(){

**return** **new** ClassRoom( callStudent() );

}

@Bean

**public** Student callStudent(){

**return** **new** Student( );

}

}

**Program2: ClassRoom.java**

package com.oracle;

public class ClassRoom {

private Student student;

public ClassRoom(Student student){

System.out.println("Inside ClassRoom constructor." );

this.student = student;

}

public void callStudent(){ student.callingStudent();

}

}

**Program3: Student.java**

**package** com.oracle;

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

**public** Student(){

System.*out*.println("Inside Student constructor." );

}

**public** **void** callingStudent(){ System.*out*.println("Inside calling Student." );

}

}

**Program4: MainApplication.java**

package com.oracle;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApplication {

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ClassRoomConfig.class);

ClassRoom cr = ctx.getBean(ClassRoom.class);

cr.callStudent();

}

}

**Note:**

After adding the cgilib.jar and asm.jar, run the MainApplication.java from eclipse, if everything is fine, it should produce the following output:

Inside Student constructor.

Inside ClassRoom constructor.

Inside calling Student.

**@Configuration and @Bean Annotations:**

This annotation is used for loading @Bean definitions from another configuration class.

**Example:**

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

The above defines the configuration file ConfigA which has the bean definition “A”.

Now, let’s have another configuration file:

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B a() {

return new A();

}

}

In this configuration file (ConfigB.java), we have imported the bean defintions from ConfigA.

In the main application, when instantiating the context, only ConfigB needs to be supplied as follows:

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

Note that we don’t need to mention ConfigA here.

**Lifecycle callbacks:**

Now, recall again the lifecycle callbacks which we had mentioned in the XML file (init-metod & destroy-method). These also can be expressed in Java based annotations.

public class A {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class ApplicationConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public A a() {

return new A();

}

}

**Bean Scope:**

Bean scope also can be expressed in Java based annotations. The default is singleton, but you can override as follows:

@Configuration

public class ApplicationConfig {

@Bean

@Scope("prototype")

public A a() {

return new A();

}

}