1. **What is Spring?**
2. Spring is an open source development framework for Enterprise Java.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform.

Spring framework targets to make Java EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

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

1. **What are benefits of Spring Framework?**
2. **Lightweight:** Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.

**Inversion of control (IOC):** Loose coupling is achieved in Spring, with the Inversion of Control technique. The objects give their dependencies instead of creating or looking for dependent objects.

**Aspect oriented (AOP):** Spring supports Aspect oriented programming and separates application business logic from system services.

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

**MVC Framework:** Spring’s web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks.

**Transaction Management:** Spring provides a consistent transaction management interface that can scale down to a local transaction and scale up to global transactions (JTA).

**Exception Handling:**  Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions.

1. **Which are the Spring framework modules?**
2. The basic modules of the Spring framework are:

### Core Container

The [Core Container](http://docs.spring.io/spring-framework/docs/3.0.x/reference/beans.html#beans-introduction) consists of the Core, Beans, Context, and Expression Language modules.

The [Core and Beans](http://docs.spring.io/spring-framework/docs/3.0.x/reference/beans.html#beans-introduction) modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The [Context](http://docs.spring.io/spring-framework/docs/3.0.x/reference/beans.html#context-introduction) module builds on the solid base provided by the [Core and Beans](http://docs.spring.io/spring-framework/docs/3.0.x/reference/beans.html#beans-introduction) modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The [Expression Language](http://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html) module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

### Data Access/Integration

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

The [JDBC](http://docs.spring.io/spring-framework/docs/3.0.x/reference/jdbc.html#jdbc-introduction) module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The [ORM](http://docs.spring.io/spring-framework/docs/3.0.x/reference/orm.html#orm-introduction) module provides integration layers for popular object-relational mapping APIs, including [JPA](http://docs.spring.io/spring-framework/docs/3.0.x/reference/orm.html#orm-jpa), [JDO](http://docs.spring.io/spring-framework/docs/3.0.x/reference/orm.html#orm-jdo), [Hibernate](http://docs.spring.io/spring-framework/docs/3.0.x/reference/orm.html#orm-hibernate), and [iBatis](http://docs.spring.io/spring-framework/docs/3.0.x/reference/orm.html" \l "orm-ibatis" \o "13.6 iBATIS SQL Maps). Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

The [OXM](http://docs.spring.io/spring-framework/docs/3.0.x/reference/oxm.html) module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

The Java Messaging Service ([JMS](http://docs.spring.io/spring-framework/docs/3.0.x/reference/jms.html)) module contains features for producing and consuming messages.

The [Transaction](http://docs.spring.io/spring-framework/docs/3.0.x/reference/transaction.html) module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (plain old Java objects).

### Web

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

Spring's 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. It also contains the web-related parts of Spring's remoting support.

The Web-Servlet module contains Spring's model-view-controller ([MVC](http://docs.spring.io/spring-framework/docs/3.0.x/reference/mvc.html#mvc-introduction)) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

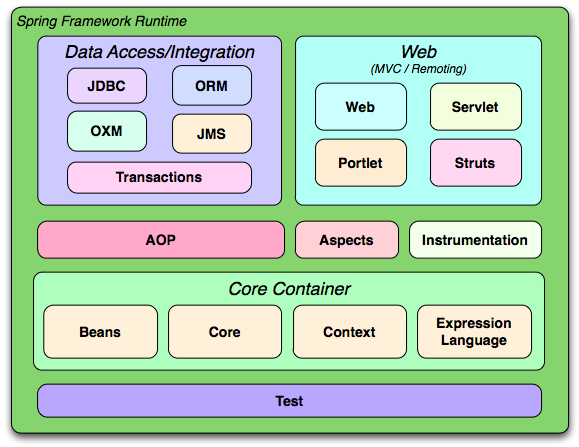
### AOP and Instrumentation

Spring's [AOP](http://docs.spring.io/spring-framework/docs/3.0.x/reference/aop.html#aop-introduction) module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate Aspects module provides integration with AspectJ.

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

<http://docs.spring.io/spring-framework/docs/3.0.x/reference/images/spring-overview.png>



1. **What is Inversion of Control?**
2. Inversion of Control (IoC) is a general concept, **this concept says that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file**. A container (the IOC container) is then responsible for hooking it all up.

It can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

<http://www.codeproject.com/Articles/380748/Inversion-of-Control-Overview-with-Examples>

1. **What are the benefits of IOC?**
2. The main benefits of IOC or dependency injection are:

* It minimizes the amount of code in your application.
* It makes your application easy to test as it doesn't require any singletons or JNDI lookup mechanisms in your unit test cases.
* Loose coupling is promoted with minimal effort and least intrusive mechanism.
* IOC containers support eager instantiation and lazy loading of services.

1. **What is Spring IoC container?**
2. The Spring IoC creates (instantiates) the objects, configure and assemble(wire) the objects, and manage their complete lifecycle from creation till destruction. The IoC container gets information from the XML file and works accordingly.

The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**
3. **What is dependency injection?**

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. To understand the DI better, Let's understand the Dependency Lookup (DL) first:

### Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

A obj = new AImpl();

In such way, we get the resource (instance of A class) directly by new keyword. Another way is factory method:

A obj = A.getA();

This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

1. Context ctx = new InitialContext();
2. Context environmentCtx = (Context) ctx.lookup("java:comp/env");
3. A obj = (A)environmentCtx.lookup("A");

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

### Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

* **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
* **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

### Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

**class** Employee{

Address address;

Employee(Address address){

**this**.address=address;

}

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

**this**.address=address;

}

}

In such case, instance of Address class is provided by external source such as XML file either by constructor or setter method.

### Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

[**http://www.javatpoint.com/dependency-injection-in-spring**](http://www.javatpoint.com/dependency-injection-in-spring)

1. **What are the different types of IoC (dependency injection)?**
2. Types of IoC are:

* **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.
* **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:** There are 3 types of dependency injections:-

1. Constructor Injection- Spring supports it.

2. Setter Injection - Spring supports it.

3. Interface Injection - Spring does not support it.

Spring supports only constructor and setter based injection.

1. **What is Spring configuration file?**
2. Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

|  |  |
| --- | --- |
| **Properties** | **Description** |
| class | This attribute is mandatory and specify the bean class to be used to create the bean. |
| name | This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). |
| scope | This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter. |
| constructor-arg | This is used to inject the dependencies and will be discussed in next chapters. |
| property | This is used to inject the dependencies and will be discussed in next chapters. |
| autowiring mode | This is used to inject the dependencies and will be discussed in next chapters. |
| lazy-initialization mode | A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup. |
| initialization method | A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter. |
| destruction method | A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter. |

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

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

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

</bean>

<!-- Definition for textEditor bean, injecting object as constructor arg -->

<bean id=*"textEditor"* class=*"com.ridkar.bean.TextEditor"*>

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

</bean>

<!-- Definition for spellChecker bean, injecting string as constructor arg -->

<bean id=*"spellChecker"* class=*"com.ridkar.bean.SpellChecker"*>

<constructor-arg name=*"constMessage"* value=*"Inside SpellChecker constructor."* />

<property name=*"methodMessage"* value=*"Inside checkSpelling."* />

</bean>

</beans>

<!-- app-config.xml -->

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

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

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

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

xsi:schemaLocation="...">

<bean id="transferService"

class="com.bank.service.internal.DefaultTransferService">

<constructor-arg ref="accountRepository"/>

<constructor-arg ref="feePolicy"/>

</bean>

<bean id="accountRepository"

class="com.bank.repository.internal.JdbcAccountRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="feePolicy"

class="com.bank.service.internal.ZeroFeePolicy"/>

<beans profile="dev">

<jdbc:embedded-database id="dataSource">

<jdbc:script

location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script

location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

<beans profile="default">

<jdbc:embedded-database id="dataSource">

<jdbc:script

location="classpath:com/bank/config/sql/schema.sql"/>

</jdbc:embedded-database>

</beans>

</beans>

1. **What is profie?**

[**http://www.captaindebug.com/2012/08/using-spring-profiles-in-xml-config.html#.VvlIOUcyZyU**](http://www.captaindebug.com/2012/08/using-spring-profiles-in-xml-config.html#.VvlIOUcyZyU)

[**http://hmkcode.com/spring-profile-xml-configuration/**](http://hmkcode.com/spring-profile-xml-configuration/)

1. **Constructor-based or setter-based DI?**

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use **constructors for *mandatory dependencies*** and **setter methods or configuration methods for *optional dependencies*.** Note that use of the [@Required](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-required-annotation) annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not null. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code smell*, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through [JMX MBeans](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jmx.html) is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

1. **What is a Circular dependency?**
   1. If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a **BeanCurrentlyInCreationException**.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

1. **What happens when suppose singleton bean A needs to use non-singleton (prototype) bean B?**

A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the ApplicationContextAware interface, and by making a getBean("B") call to the container ask for (a typically new) bean B instance every time bean A needs it.

*// a class that uses a stateful Command-style class to perform some processing*

**package** fiona.apple;

*// Spring-API imports*

**import** org.springframework.beans.BeansException;

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

**import** org.springframework.context.ApplicationContextAware;

**public** **class** CommandManager **implements** ApplicationContextAware {

**private** ApplicationContext applicationContext;

**public** Object process(Map commandState) {

*// grab a new instance of the appropriate Command*

Command command = createCommand(); 2.

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

**return** **this**.applicationContext.getBean("command", Command.**class**); 3.

}

**public** **void** setApplicationContext(

ApplicationContext applicationContext) **throws** BeansException { 1.

**this**.applicationContext = applicationContext;

}

}

1. **What are types of IoC containers? Explain them.**
2. There are two types of IoC containers:

* **Bean Factory container:** This is the simplest container providing basic support for DI .The BeanFactory is usually preferred where the resources are limited like mobile devices or applet based applications. This is defined by the org.springframework.beans.factory.BeanFactory interface.
* **Spring ApplicationContext Container:** This container adds more enterprise-specific functionality such as the ability to
* resolve textual messages from a properties file
* publish application events to interested event listeners
* AOP
* Internationalization etc

This container is defined by the org.springframework.context.ApplicationContext interface.

1. **What is the difference between Bean Factory and ApplicationContext?**

This one is very popular spring interview question and often asks in entry level interview. ApplicationContext is the preferred way of using spring because of functionality provided by it and interviewer wanted to check whether you are familiar with it or not.

|  |  |
| --- | --- |
| **ApplicationContext.** | **BeanFactory** |
| Here we can have more than one config files possible | In this only one config file or .xml file |
| Application contexts can publish events to beans that are registered as listeners | Doesn’t support. |
| Application contexts provide a means for resolving text messages, including support for i18n of those messages. | It’s not |
| Support application life-cycle events, and validation. | Doesn’t support. |
| Supports  many enterprise services such JNDI access, EJB integration, remoting | Doesn’t support. |

1. **Give an example of BeanFactory implementation.**
2. The most commonly used BeanFactory implementation is the**XmlBeanFactory** class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

Deprecated as of Spring 3.1 in favor of DefaultListableBeanFactory and XmlBeanDefinitionReader

Before:

public class MainApp {

public static void main(String[] args) {

XmlBeanFactory factory = new XmlBeanFactory(new ClassPathResource("Beans.xml"));

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

obj.getMessage();

}

}

After:

public class MainApp {

public static void main(String[] args) {

// Resource resource = new ClassPathResource("applicationBean.xml");

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

BeanDefinitionReader reader = new XmlBeanDefinitionReader(beanFactory);

// reader.loadBeanDefinitions(resource);

reader.loadBeanDefinitions(new ClassPathResource("applicationBean.xml"));

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

obj.getMessage();

}

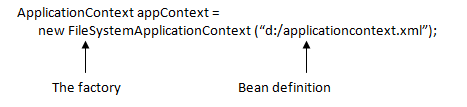
}

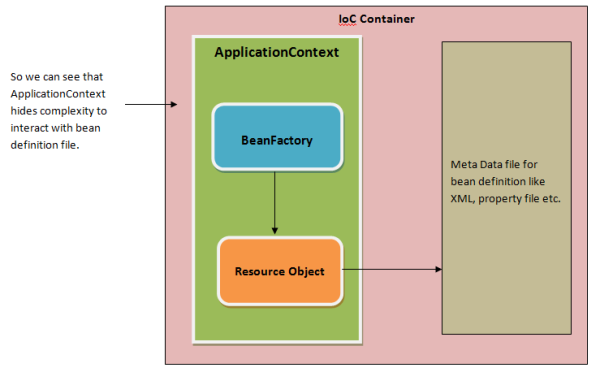
1. **What are the common implementations of the ApplicationContext?**
2. The three commonly used implementation of 'Application Context' are:

* **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.
* **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.
* **WebXmlApplicationContext:** This container loads the XML file with definitions of all beans from within a web application.
* **GenericXmlApplicationContext:** Spring now recommend that you use the GenericApplicationContext class instead of ClassPathXmlApplicationContext and FileSystemXmlApplicationContext as this provides additional flexibility. For example, in the code above, I’ve used GenericApplicationContext’s load(...) method to load a number of configuration files using a wild card:

    ctx.load("\*-profile.xml");

In case of ApplicationContext, the resource object is implemented internally, so as use we don’t need of resource object.

[](https://techythought.files.wordpress.com/2013/01/context4.png)

[](https://techythought.files.wordpress.com/2013/01/cntext2.png)<https://techythought.wordpress.com/category/spring/>

<http://docs.spring.io/autorepo/docs/spring/1.2.9/reference/beans.html>

1. **What are Spring beans?**
2. The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is **instantiated, assembled**, and otherwise **managed by a Spring IoC container**. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.
3. **What does a bean definition contain?**
4. The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

1. **How do you provide configuration metadata to the Spring Container?**
2. There are following three important methods to provide configuration metadata to the Spring Container:

* XML based configuration file.
* Annotation-based configuration
* Java-based configuration

1. **How do add a bean in spring application?**
2. Check the following 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">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

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

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

1. **How do you define a bean scope?**
2. When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be prototype. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be singleton.

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld" scope="singleton">

</bean>

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld" scope="prototype">

</bean>

1. **What bean scopes does Spring support? Explain them.**
2. The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

* **singleton**: This scopes the bean definition to a single instance per Spring IoC container.
* **prototype**: This scopes a single bean definition to have any number of object instances.
* **request**: This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
* **session**: This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
* **global-session**: This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

1. **What is default scope of bean in Spring framework?**
   1. The **default** scope of bean is **Singleton** for Spring framework.
2. **Are Singleton beans thread safe in Spring Framework?** 
   1. No, singleton beans are not thread-safe in Spring framework.
3. **Explain Bean lifecycle in Spring framework?**
   1. Following is sequence of a bean lifecycle in Spring:

* **Instantiate** - First the spring container finds the bean's definition from the XML file and instantiates the bean.
* **Populate properties** - Using the dependency injection, spring populates all of the properties as specified in the bean definition..
* **Set Bean Name** - If the bean implements BeanNameAware interface, spring passes the bean's id to setBeanName() method.
* **Set Bean factory** - If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
* **Pre Initialization** - Also called postprocess of bean. If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization() method.
* **Initialize beans** - If the bean implements IntializingBean,its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
* **Post Initialization** - If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
* **Ready to use** - Now the bean is ready to use by the application.
* **Destroy** - If the bean implements DisposableBean , it will call the destroy() method .

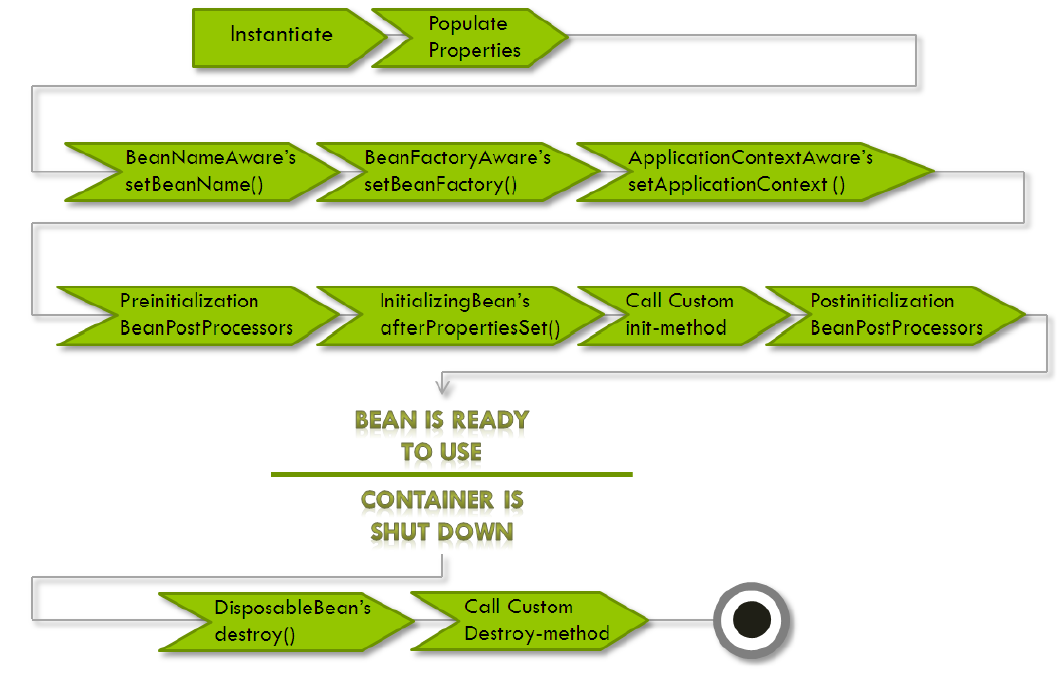
<http://www.journaldev.com/2637/spring-bean-life-cycle>

1. **How BeanPostProcessor works?**

The initialization here refers to calling an init method of your triangle bean, so the before and post calls wrap the call on the init method of the bean. Prior to that, the constructor and setter injections are already performed, that is why the bean has actual values.

<http://www.tutorialspoint.com/spring/spring_bean_post_processors.htm>

<https://dzone.com/articles/interacting-newly-created-bean>



Jul 01, 2016 9:44:07 PM org.springframework.core.SpringProperties <clinit>

INFO: Found 'spring.properties' file in local classpath

Jul 01, 2016 9:44:07 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@3dc049d: startup date [Fri Jul 01 21:44:07 IST 2016]; root of context hierarchy

Jul 01, 2016 9:44:07 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [spring.xml]

Jul 01, 2016 9:44:08 PM org.springframework.beans.factory.config.PropertyPlaceholderConfigurer loadProperties

INFO: Loading properties file from class path resource [spring.properties] **BeanFactoryPostProcessor:** in postProcessBeanFactory

**Construtor:** BeanLifeCycle constructor is created

**BeanNameAware:** In setBeanName with bean state: I am a test bean to demo bean life cycle

**ApplicationContextAware:** In setApplicationContext with bean state: I am a test bean to demo bean life cycle

**BeanPostProcessor:** In postProcessBeforeInitialization name: beanLifeCycle

**InitializingBean:** In afterPropertiesSet with bean state: I am a test bean to demo bean life cycle

**default-init-method:** In \*\*myInit\*\* with bean state: I am a test bean to demo bean life cycle

**BeanPostProcessor:** In postProcessAfterInitialization name: beanLifeCycle

**Bean Method show():** I am a test bean to demo bean life cycle

**DisposableBean:** In destroy with bean state: I am a test bean to demo bean life cycle

**default-destroy-method:** In \*\*myDestroy\*\* with bean state: I am a test bean to demo bean life cycle

1. **What are inner beans in Spring?**
   1. A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean. An inner bean definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag. Inner beans are always anonymous and they are always scoped as prototypes.

[**http://www.tutorialspoint.com/spring/spring\_injecting\_inner\_beans.htm**](http://www.tutorialspoint.com/spring/spring_injecting_inner_beans.htm)

<?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 textEditor bean using inner bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker">

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>

</property>

</bean>

</beans>

1. **How can you inject Java Collection in Spring?**
2. Spring offers four types of collection configuration elements which are as follows:

* **<list>**: This helps in wiring i.e. 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.

<?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 javaCollection -->

<bean id="javaCollection" class="com.tutorialspoint.JavaCollection">

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

<property name="addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

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

<property name="addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

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

<property name="addressMap">

<map>

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

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

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

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

</map>

</property>

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

<property name="addressProp">

<props>

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

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

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

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

</props>

</property>

</bean>

</beans>

1. **What is bean auto wiring?**
   1. The Spring container is able to autowire relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory without using <constructor-arg> and <property> elements.
2. **What are different Modes of auto wiring?**
3. The autowiring functionality has five modes which can be used to instruct Spring container to use autowiring for dependency injection:

* **no**: This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter.
* **byName**: Autowiring 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.
* **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**: When we use autowire byName or byType, default constructor is used. That’s why we have explicitly defined the default constructor for the EmployeeService bean.

By default spring bean autowiring is turned off. Spring bean autowire default value is “default” that means no autowiring is to be performed. autowire value “no” also have the same behavior.

Following is the configuration file Beans.xml in 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 textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

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

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

But if you are going to use autowiring 'byName', then your XML configuration file will become as follows:

<?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 textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor"

autowire="byName">

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

[**http://www.journaldev.com/2623/spring-bean-autowire-by-name-type-constructor-autowired-and-qualifier-annotations-example**](http://www.journaldev.com/2623/spring-bean-autowire-by-name-type-constructor-autowired-and-qualifier-annotations-example)

1. **What are the limitations with autowiring?**
2. Limitations of autowiring are:

* **Overriding possibility**: You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
* **Primitive data types**: You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
* **Confusing nature**: Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

1. **What is the precedence of Spring beans definitions?**

* Beans are registered in the orders that are found in xml definition file.
* Scanned beans are registered at point that the xml tag is found but scanned beans cannot override previously registered bean definitions.
* Xml beans definitions can override any previously bean definition if DefaultListableBeanFactory.allowBeanDefinitionOverriding is true (by default).

So XML Wins.

If you put the component-scan tag first, xml beans will override scanned ones. If you put it last, scanned beans will be ignored.

[**http://stackoverflow.com/questions/15233766/what-is-the-precedence-of-spring-beans-definitions**](http://stackoverflow.com/questions/15233766/what-is-the-precedence-of-spring-beans-definitions)

1. **Can you inject null and empty string values in Spring?**
2. Yes.

<bean id="defaultMongoTypeMapper"

class="org.springframework.data.mongodb.core.convert.DefaultMongoTypeMapper">

<constructor-arg name="typeKey"><null /></constructor-arg>

</bean>

<bean id="myConverter"

class="com.mkyong.convert.MoneyConverter">

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

</bean>

1. **What is Annotation-based container configuration?**
2. An alternative to XML setups is provided by annotation-based configuration which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
3. **How do you turn on annotation wiring?**
4. 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 by configuring <context:annotation-config/>

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

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

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

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

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

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

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

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

<context:annotation-config />

<bean id=*"toi"* class=*"com.ridkar.bean.NewsRack"*>

<property name=*"name"* value=*"Times of India"*></property>

<!-- <property name="news" ref="news"></property> -->

</bean>

<bean id=*"news"* class=*"com.ridkar.bean.News"*>

<property name=*"news"* value=*"News is Zindabad"*></property>

<property name=*"headline"* value=*"Here are the headlines"*></property>

</bean>

</beans>

1. **What is** [**difference between <context:annotation-config> vs <context:component-scan>**](http://stackoverflow.com/questions/7414794/difference-between-contextannotation-config-vs-contextcomponent-scan)**?**

## http://howtodoinjava.com/spring/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/

## context:component-scan

This element has been introduced in Spring configuration from version 2.5. If you have worked with the previous versions of Spring, all the beans has to be manually configured in the XML files. There are no annotations supported in the Java beans. This will result in lot of XML code in the configuration files and every time developer has to update the XML file for configuring the new beans. **context:component-scan** element in the spring configuration file would eliminate the need for declaring all the beans in the XML files. Look at the below declaration in your spring configuration file:

|  |  |
| --- | --- |
| 1 | <context:component-scan base-package="org.controller"/> |

The above declaration in the spring application configuration file would scan the classes inside the specified package and create the beans instance. Note that it could create beans only if that class is annotated with correct annotations. The following are the annotations scanned by this element:

* **@Component**
* **@Repository**
* **@Service**
* **@Controller**

One advantage of this element is that it also resolve [@Autowired](http://www.javabeat.net/autowired-annotation-spring/) and [@Qualifier](http://www.javabeat.net/qualifier-annotation-spring/) annotations. Therefore if you declare **<context:component-scan>**, is not necessary anymore declare **<context:annotation-config>** too.

## mvc:annotation-driven

**mvc:annotation-driven** is used for enabling the Spring MVC components with its default configurations. If you don’t include mvc:annotation-driven also your MVC application would work if you have used the context:component-scan for creating the beans or defined the beans in your XML file.

But, mvc:annotation-driven does some extra job on configuring the special beans that would not have been configured if you are not using this element in your XML file.

This tag would registers the HandlerMapping and HandlerAdapter required to dispatch requests to your @Controllers. In addition, it also applies some defaults based on what is present in your classpath. Such defaults are:

* Using the Spring 3 Type ConversionService as a simpler and more robust alternative to JavaBeans PropertyEditors
* Support for formatting Number fields with @NumberFormat
* Support for formatting Date, Calendar, and Joda Time fields with @DateTimeFormat, if Joda Time is on the classpath
* Support for validating @Controller inputs with @Valid, if a JSR-303 Provider is on the classpath
* Support for reading and writing XML, if JAXB is on the classpath
* Support for reading and writing JSON, if Jackson is on the classpath

## context:annotation-config

context:annotation-config is used for activating annotations in beans already registered in the application context (no matter whether they were defined with XML or by package scanning). That means it will resolve @Autowired and @Qualifier annotations for the beans which are already created and stored in the spring container.

**context:component-scan** can also do the same job, but context:component-scan will also scan the packages for registering the beans to application context. context:annotation-config will not search for the beans registration, this will only activate the already registered beans in the context.

[**http://www.javabeat.net/spring-mvc-component-scan-annotations/**](http://www.javabeat.net/spring-mvc-component-scan-annotations/)

[**http://stackoverflow.com/questions/7414794/difference-between-contextannotation-config-vs-contextcomponent-scan**](http://stackoverflow.com/questions/7414794/difference-between-contextannotation-config-vs-contextcomponent-scan)

1. **What does @Required annotation mean?**
   1. The **@Required** annotation applies to bean property setter methods.

It indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

Otherwise the container throws a BeanInitializationException exception.

1. **What does @Autowired annotation mean?**
   1. This 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.

[**http://www.tutorialspoint.com/spring/spring\_autowired\_annotation.htm**](http://www.tutorialspoint.com/spring/spring_autowired_annotation.htm)

1. **What does @Qualifier annotation mean?**
   1. 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.

[**http://www.tutorialspoint.com/spring/spring\_qualifier\_annotation.htm**](http://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm)

1. **What are the JSR-250 Annotations? Explain them.**
2. Spring has JSR-250 based annotations which include @PostConstruct, @PreDestroy and @Resource annotations.

* @PostConstruct: This annotation can be used as an alternate of initialization callback.
* @PreDestroy: This annotation can be used as an alternate of destruction callback.
* @Resource : This annotation can be used on fields or setter methods. The @Resource annotation takes a 'name' attribute which will be interpreted as the bean name to be injected. You can say, it follows by-name autowiring semantics.

1. **What is Spring Java Based Configuration?**
2. Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.

For example: Annotation **@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.

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

You can import above Bean declaration in another Bean Declaration as follows:

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B a() {

return new A();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class 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);

}

1. **Give some annotation example. How is event handling done in Spring?**
2. Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. So if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.
3. **Describe some of the standard Spring events.**
4. Spring provides the following standard events:

* **ContextRefreshedEvent**: This event is published when the ApplicationContext is either initialized or refreshed. This can also be raised using the refresh() method on the ConfigurableApplicationContext interface.
* **ContextStartedEvent**: This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can re/start any stopped application after receiving this event.
* **ContextStoppedEvent**: This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do required housekeep work after receiving this event.
* **ContextClosedEvent**: This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
* **RequestHandledEvent**: This is a web-specific event telling all beans that an HTTP request has been serviced.

1. **What is AOP?**
2. Aspect-oriented programming, or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management. The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules.

<http://www.mkyong.com/spring/spring-aop-examples-advice/>

1. **What is Aspect?**
   1. A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).
2. **What is the difference between concern and cross-cutting concern in Spring AOP?**
   1. **Concern**: Concern is behavior which we want to have in a module of an application. Concern may be defined as a functionality we want to implement. Issues in which we are interested define our concerns.

**Cross-cutting concern**: It's a concern which is applicable throughout the application and it affects the entire application. e.g. logging , security and data transfer are the concerns which are needed in almost every module of an application, hence are cross-cutting concerns.

1. **What is Join point?**
   1. This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
2. **What is Advice?**
   1. This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.
3. **What is Pointcut?**
   1. Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.

package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class EmployeeAspect {

    @Before("execution(public String getName())")

    public void getNameAdvice(){

        System.out.println("Executing Advice on getName()");

    }

    @Before("execution(\* com.journaldev.spring.service.\*.get\*())")

    public void getAllAdvice(){

        System.out.println("Service method getter called");

    }

}

Important points in above aspect class is:

* Aspect classes are required to have @Aspect annotation.
* @Before annotation is used to create Before advice
* The string parameter passed in the @Before annotation is the Pointcut expression
* *getNameAdvice()* advice will execute for any Spring Bean method with signature public String getName(). This is a very important point to remember, if we will create Employee bean using new operator the advices will not be applied. Only when we will use ApplicationContext to get the bean, advices will be applied.
* We can use asterisk (\*) as wild card in Pointcut expressions, *getAllAdvice()* will be applied for all the classes in com.journaldev.spring.service package whose name starts with get and doesn’t take any arguments.

<http://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations-xml-configuration>

1. **What is Introduction?**
   1. An introduction allows you to add new methods or attributes to existing classes.
2. **What is Target object?**
   1. They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
3. **What is Weaving?**
   1. Weaving is the process of linking aspects with other application types or objects to create an advised object

The AspectJ weaver takes class files as input and produces class files as output. The weaving process itself can take place at one of three different times: compile-time, post-compile time, and load-time. The class files produced by the weaving process (and hence the run-time behaviour of an application) are the same regardless of the approach chosen.

* Compile-time weaving is the simplest approach. When you have the source code for an application, ajc will compile from source and produce woven class files as output. The invocation of the weaver is integral to the ajc compilation process. The aspects themselves may be in source or binary form. If the aspects are required for the affected classes to compile, then you must weave at compile-time. Aspects are required, e.g., when they add members to a class and other classes being compiled reference the added members.
* Post-compile weaving (also sometimes called binary weaving) is used to weave existing class files and JAR files. As with compile-time weaving, the aspects used for weaving may be in source or binary form, and may themselves be woven by aspects.
* Load-time weaving (LTW) is simply binary weaving defered until the point that a class loader loads a class file and defines the class to the JVM. To support this, one or more "weaving class loaders", either provided explicitly by the run-time environment or enabled through a "weaving agent" are required.

You may also hear the term "run-time weaving". We define this as the weaving of classes that have already been defined to the JVM (without reloading those classes). AspectJ 5 does not provide explicit support for run-time weaving although simple coding patterns can support dynamically enabling and disabling advice in aspects.

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the @AspectJ-style of aspect declaration.

package foo;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.util.StopWatch;

import org.springframework.core.annotation.Order;

*@Aspect*

public class ProfilingAspect {

*@Around("methodsToBeProfiled()")*

public Object profile(ProceedingJoinPoint pjp) throws Throwable {

StopWatch sw = new StopWatch(getClass().getSimpleName());

try {

sw.start(pjp.getSignature().getName());

return pjp.proceed();

} finally {

sw.stop();

System.out.println(sw.prettyPrint());

}

}

@Pointcut("execution(public \* foo..**.**(..))")

public void methodsToBeProfiled(){}

}

We will also need to create an ' META-INF/aop.xml' file, to inform the AspectJ weaver that we want to weave our ProfilingAspect into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called ' META-INF/aop.xml' is standard AspectJ.

**<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">**

<aspectj>

<weaver>

<!-- only weave classes in our application-specific packages -->

<include within="foo.\*"/>

</weaver>

<aspects>

<!-- weave in just this aspect -->

<aspect name="foo.ProfilingAspect"/>

</aspects>

</aspectj>

Now to the Spring-specific portion of the configuration. We need to configure a LoadTimeWeaver (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more ' META-INF/aop.xml' files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

<?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">

<!-- a service object; we will be profiling its methods -->

<bean id="entitlementCalculationService"

class="foo.StubEntitlementCalculationService"/>

<!-- this switches on the load-time weaving -->

**<context:load-time-weaver/>**

</beans>

.

1. **What are the different points where weaving can be applied?**
   1. Weaving can be done at compile time, load time, or at runtime.
2. **What are the types of advice?**
3. Spring aspects can work with five kinds of advice mentioned below:

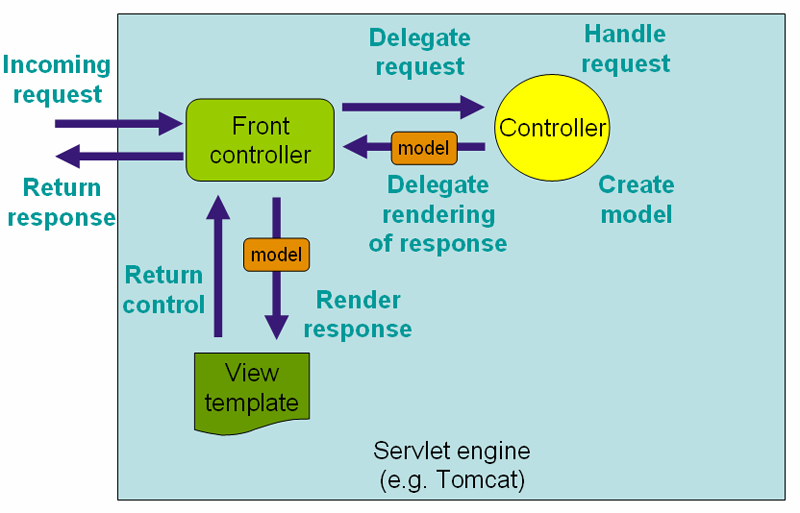
* **before**: Run advice before the a method execution.
* **after**: Run advice after the a method execution regardless of its outcome.
* **after-returning**: Run advice after the a method execution only if method completes successfully.
* **after-throwing**: Run advice after the a method execution only if method exits by throwing an exception.
* **around**: Run advice before and after the advised method is invoked.

1. **What is XML Schema based aspect implementation?**
   1. Aspects are implemented using regular classes along with XML based configuration.
2. **What is @AspectJ? based aspect implementation?**
3. AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.
4. **How JDBC can be used more efficiently in spring framework?**
5. JDBC can be used more efficiently with the help of a template class provided by spring framework called as JDBC Template.
6. **How JDBC Template can be used?**
7. With use of Spring JDBC framework the burden of resource management and error handling is reduced a lot. So it leaves developers to write the statements and queries to get the data to and from the database. JDBC Template provides many convenience methods for doing things such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.
8. **What are the types of the transaction management Spring supports?**
9. Spring supports two types of transaction management:

* **Programmatic transaction management:** This means that you have managed the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
* **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

1. **Which of the above transaction management type is preferable?**
   1. Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code.
2. **What is Spring MVC framework?**
3. The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

<http://javabeginnerstutorial.com/spring-framework-tutorial/developing-a-spring-3-framework-mvc-application-step-by-step-tutorial/>



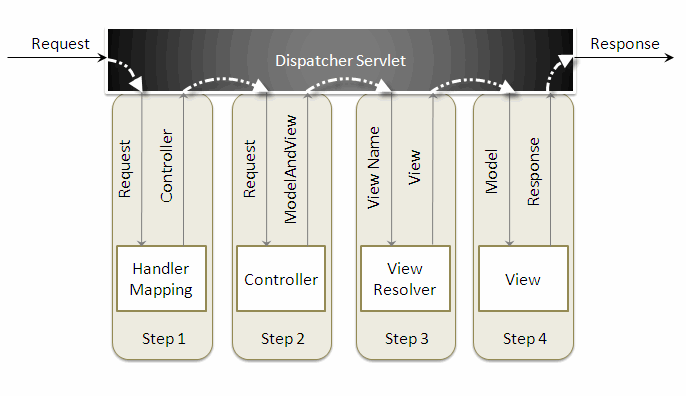
1. **What is a DispatcherServlet?**
2. The Spring Web MVC framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

* After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
* The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
* The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.



1. **What is WebApplicationContext ?**
2. In the web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext.

The *WebApplicationContext* is an extension of the plain *ApplicationContext* that has some extra features necessary for web applications. It differs from a normal *ApplicationContext* in that it is capable of resolving themes, and that it knows which servlet it is associated with.



1. **List down spring annotations.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Annotation Type** | **Annotation** | **Use** | **Description** |
| Context Configuration | @Autowired | Constructor, Field, Method | Declares a constructor, field, setter method, or configuration method to be autowired by type. Items annotated with @Autowired do not have to be public. |
| Context Configuration | @Configurable | Type | Used with <context:springconfigured> to declare types whose properties should be injected, even if they are not instantiated by Spring. Typically used to inject the properties of domain objects. |
| Context Configuration | @Order | Type, Method, Field | Defines ordering, as an alternative to implementing the org. springframework.core.Ordered interface. |
| Context Configuration | @Qualifier | Field, Parameter, Type, Annotation Type | Guides autowiring to be performed by means other than by type. |
| Context Configuration | @Required | Method (setters) | Specifies that a particular property must be injected or else the configuration will fail. |
| Context Configuration | @Scope | Type | Specifies the scope of a bean, either singleton, prototype, request, session, or some custom scope. |
| Stereotyping | @Component | Type | Generic stereotype annotation for any Spring-managed component. |
| Stereotyping | @Controller | Type | Stereotypes a component as a Spring MVC controller. |
| Stereotyping | @Repository | Type | Stereotypes a component as a repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions. |
| Stereotyping | @Service | Type | Stereotypes a component as a service. |
| MVC | @Controller | Type | Stereotypes a component as a Spring MVC controller. |
| MVC | @InitBinder | Method | Annotates a method that customizes data binding. |
| MVC | @ModelAttribute | Parameter, Method | When applied to a method, used to preload the model with the value returned from the method. When applied to a parameter, binds a model attribute to the parameter. table |
| MVC | @RequestMapping | Method, Type | Maps a URL pattern and/or HTTP method to a method or controller type. |
| MVC | @RequestParam | Parameter | Binds a request parameter to a method parameter. |
| MVC | @SessionAttributes | Type | Specifies that a model attribute should be stored in the session. |

1. **What are the advantages of Spring MVC over Struts MVC ?**
2. Following are some of the advantages of Spring MVC over Struts MVC:

* Spring's MVC is very versatile and flexible based on interfaces but Struts forces Actions and Form object into concrete inheritance.
* Spring provides both interceptors and controllers, thus helps to factor out common behavior to the handling of many requests.
* Spring can be configured with different view technologies like Freemarker, JSP, Tiles, Velocity, XLST etc. and also you can create your own custom view mechanism by implementing Spring View interface.
* In Spring MVC Controllers can be configured using DI (IOC) that makes its testing and integration easy.
* **Web tier of Spring MVC is easy to test than Struts web tier, because of the avoidance of forced concrete inheritance and explicit dependence of controllers on the dispatcher servlet.**
* **Struts force your Controllers to extend a Struts class but Spring doesn't, there are many convenience Controller implementations that you can choose to extend.**
* In Struts, Actions are coupled to the view by defining ActionForwards within a ActionMapping or globally. SpringMVC has HandlerMapping interface to support this functionality.
* With Struts, validation is usually performed (implemented) in the validate method of an ActionForm. In SpringMVC, validators are business objects that are NOT dependent on the Servlet API which makes these validators to be reused in your business logic before persisting a domain object to a database.

1. **Explain the @Controller annotation.**
2. The *@Controller* annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API.
3. **Explain @RequestMapping annotation.**
4. *@RequestMapping* annotation is used to map a URL to either an entire class or a particular handler method.
5. **What are the ways to access Hibernate by using Spring?**
6. There are two ways to access hibernate using spring:

* Inversion of Control with a Hibernate Template and Callback.
* Extending HibernateDAOSupport and Applying an AOP Interceptor node.

1. **What are ORM's Spring supports?**
2. Spring supports the following ORM's :

* Hibernate
* iBatis
* JPA (Java Persistence API)
* TopLink
* JDO (Java Data Objects)
* OJB

Spring alias, id ref

Spring collections merge