**What is Spring Framework?**

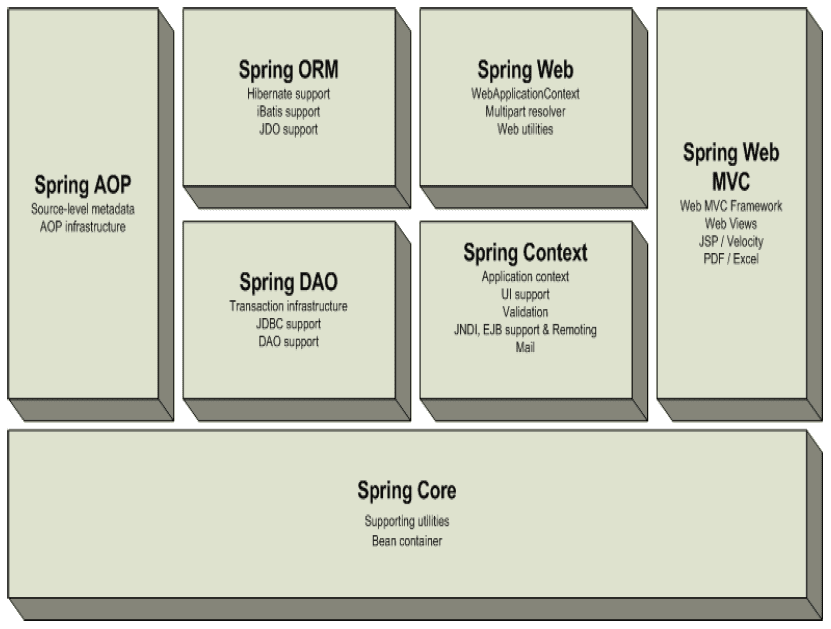
**Spring Framework** is one of the most popular Java-based application frameworks. Spring framework is an open-source java platform, and it was created by “Rod Johnson” in 2003.

Spring is a complete and modular framework. It means we can use Spring framework for all layer implementations for a real-time application, or we can use spring for the development of a particular layer of a real-time application, unlike struts [only for front end related] and hibernate [only for database related], but with spring we can develop all layers.

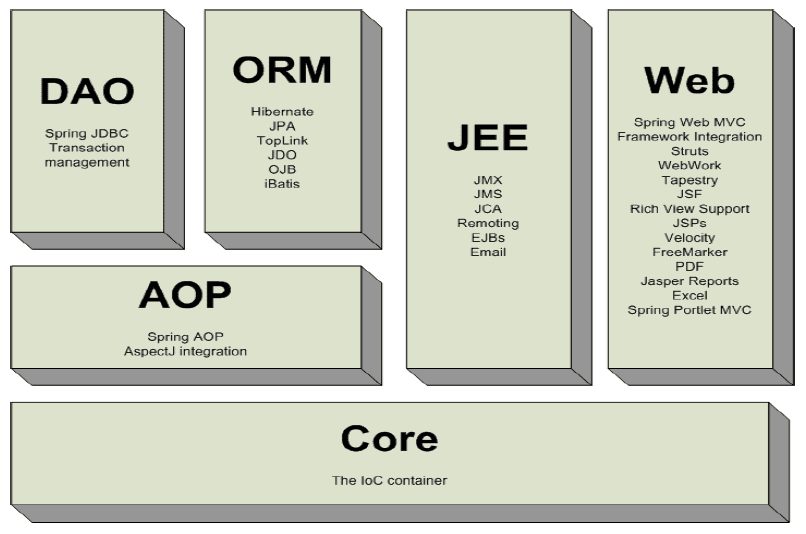
**Spring framework** can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, EJB, JSF, etc. Using web frameworks, we can develop web applications. Using ORM frameworks, we can develop persistence logic. But using the spring framework, we can develop all kinds of logic like business logic, persistence logic, integration logic, presentation logic, etc.

Using the spring framework, we can develop all kinds of applications like standalone, two-tire, web, distributed, and enterprise.

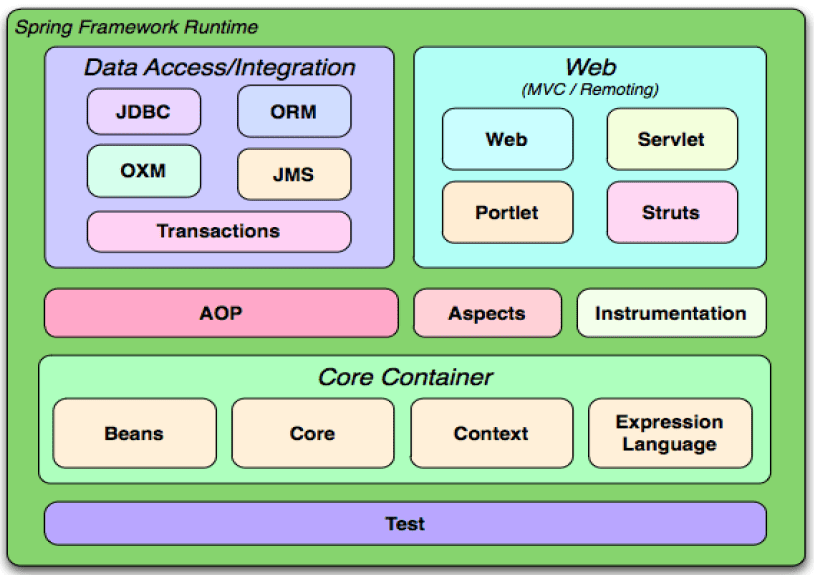
Spring 1.X contains 7 modules, spring 2.X contains 6 modules, and spring 3.X contains 20 modules, but they are grouped into 6 major categories. Spring 1.X overview diagram is as follows



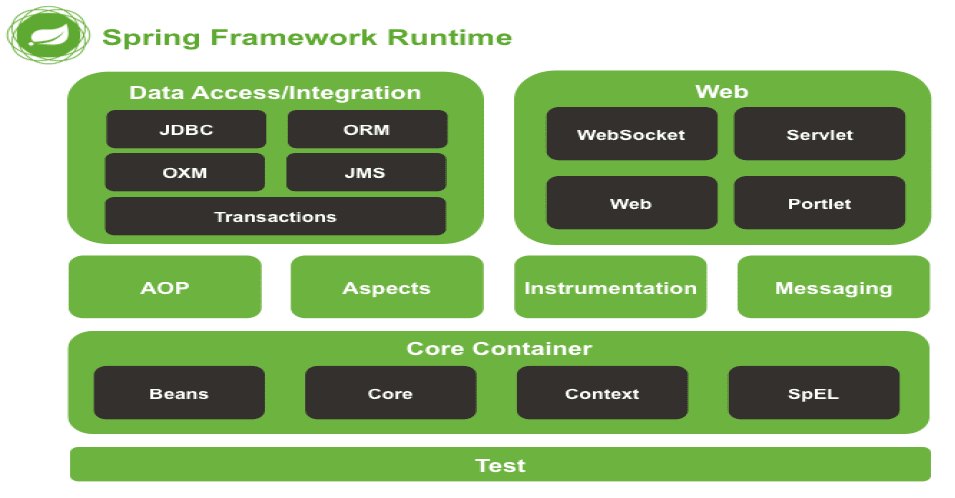
Spring 2.X overview diagram is as follows,



Spring 3.X overview diagram is as follows,



Spring 4.X overview diagram is as follows,



**Core Module**

The spring core module is fundamental in Spring Framework. It has provided a basic foundation for all other spring framework modules. We can use this module to prepare Standalone Applications directly. This module can provide the features like IOC Containers, Beans, Dependency Injection, etc.

**AOP Module [Aspect Oriented Programming]**

In general, if we prepare enterprise applications using only Object Orientation, we have to provide both business logic and Services like Transactions, JMS, JAAS, etc., in a combined manner. It will provide a tightly coupled design, and it will provide less shareability and less reusability.

To improve shareability and Reusability in the above context, we have to provide a loosely coupled design. To get a loosely coupled design, we have to apply Aspect-Oriented Programming. In Aspect-Oriented Programming, we will declare every service as an aspect and inject these aspects in Business Logic at runtime.

**JDBC/DAO Modules**

The main intention of this module is to interact with the database from the Spring application to perform database operations with the JDBC Persistence mechanism. JDBC/DAO modules can abstract common JDBC implementation to simplify Database interaction from spring applications by providing template classes.

In JDBC, if we want to interact with the database, we must use the following steps.

1. Load And Register Driver

Class.forName("oracle.jdbc.OracleDriver");

2. Establish Connection between Java application and Database.

Connction conn = DriverManager.getConnection("jdbc: oracle:thin:@localhost:1521:xe", "ashok", "ashok");

3. Create Statement/PreparedStatement/CallableStatement as per the requirement.

Statement st = conn.createStatement();

4. Write and Execute SQL Queries

ResultSet rs = st.executeQuery("select \* from emp");

5. Close the Connection

conn.close();

In the above JDBC Steps, Load and Register Driver, Establish Connection and Create Statement and close the Connection are common in all the JDBC applications. Here, the Spring JDBC/DAO modules can abstract the commonly used steps and give the developers an option to provide variable parts, that is, Executing SQL queries.

Spring DAO and JDBC modules have their own Exception Classes hierarchy to expose the exception details. Spring DAO/JDBC modules convert the JDBC generates checked exceptions to Spring defined Unchecked Exceptions using Exceptions Re-Throwing Mechanism.

**ORM Module [Object-Relational Mapping]**

The process of mapping Java class with a Database table, member variables with table columns, and making that java class objects represent Database table rows by synchronizing between them is called ORMapping.

ORM has defined a set of rules and regulations to provide a mapping between the Object-Oriented Data Model and Relational Data Model to achieve data persistency.

E.g. Hibernate, JPA, Toplink etc.

To prepare the Hibernate Application then we have to use the following instructions.

1. Create Configuration class object.

Configuration cfg = new Configuration();

cfg.configure();

2. Create SessionFactory class object

SessionFactory sf = cfg.buildSessionFactory();

3. Create Session object

Session s = sf.openSession();

4. Perform Persistence operations

Object obj = s.load("com.ashok.hibernate.model.Employee.class", 111);

5. Close SessionFactory and Configuration

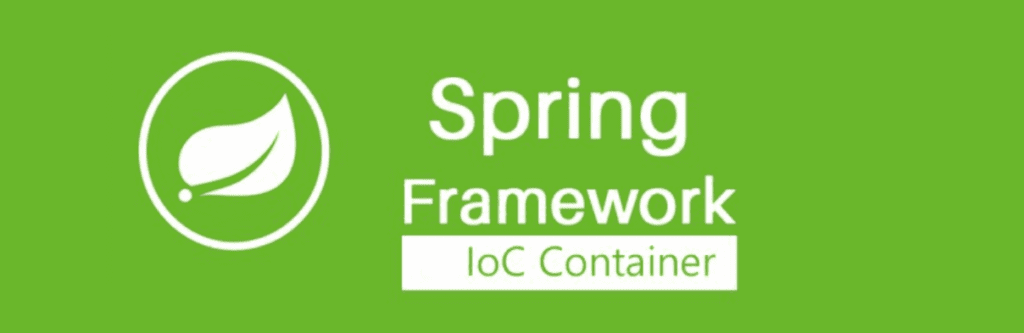
sf.close();

cfg.close();

In the above Hibernate Application steps, Create Configuration class object, Create SessionFactory object, create a Session object, and close SessionFactory and Configuration steps are common in all hibernate applications. In this case, the Spring ORM module can abstract all the common instructions to simplify Data persistence in enterprise applications.

##### Spring IOC (Inversion Of Control) Containers

we are going to discuss the Spring IOC containers. The main intention of Spring IOC Containers is to read bean configurations from a configuration file, create Bean objects and Provide bean Objects to Spring applications.



There are two types of IOC Containers in the Spring framework.

1. BeanFactory
2. ApplicationContext

###### ****1. BeanFactory****

* It is the fundamental or Base IOC container provided by Spring Framework to manage bean objects.
* BeanFactory IOC container will provide basic functionalities to the spring framework by creating maintaining beans objects as per the beans configuration details in the spring beans configuration file.
* To represent BeanFactory IOC Container, Spring framework has provided an interface in the form of “org.springframework. beans.factory.BeanFactory”.
* For the BeanFactory interface, Spring Framework has provided an implementation class in the form of org.springframework. beans.factory.xml.XmlBeanFactory.
* If we want to use BeanFactory IOC Container in Spring applications, we have to use the following steps.
  + Create Resource Object
  + Create BeanFactory object
  + Get Bean and access Business Method.

###### ****1. Create Resource**** ****Object****

* A resource is an object in Spring Framework. It can represent all bean configuration details which we provided in beans configurations details.
* To represent Resource object, Spring Framework has provided a predefined interface in the form of “org.springframework. core.io.Resource”
* For the Resource interface, Spring Framework has provided the following implementation classes.

**org.springframework.core.io.ByteArrayResource:** It can represent all the beans configuration details available in the form of byte [].

**org.springframework.core.io.FileSystemResource:** It can get all the beans configuration details available in the form of a file in our system hard disk.

**org.springframework.core.io.ClassPathResource:** It can get all the beans configuration details that exist at the “classpath” environment variable referred locations.

**org.springframework.core.io.InputStreamResource:** It can get all the beans configurations which are existed in the form of InputStream.

**org.springframework.core.io.UrlResource:** It can get all the beans configuration details that exist at a particular URL in the network.

**org.springframework.web.context.support.ServletContextResource:** It can get all the beans configuration details which are existed in ServletContext. It will be used in spring web applications.

**org.apringframework.web.portlet.context.PortletContextResource:** It can get all the beans configuration details which are existed in PortletContext. It will be used in spring web applications designed based on portlets.

E.g

Resource res = new ClassPathResource("beans.xml");

###### ****2. Create BeanFactory Object****

To create an XmlBeanFactory class object, we have to use the following constructor.

public XmlBeanFactory(Resource res)

E.g

BeanFactory factory = new XmlBeanFactrory(res);

###### ****3. Get Bean object from BeanFactory and access business method****

To get Bean object from BeanFactory we have to use the following method.

public Object getBean(String id\_Name)

E.g

Employee emp = (Employee) factory.getBean("employee");

**Note**

BeanFactory is deprecated in Spring3.x version.

E.g

**Bean Class**

package com.ashok.spring.core.ioc.beanfactory.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

private String empName;

private String empId;

private String empAddress;

private double salary;

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getEmpAddress() {

return empAddress;

}

public void setEmpAddress(String empAddress) {

this.empAddress = empAddress;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Employee [empName=" + empName + ", empId=" + empId + ", empAddress=" + empAddress + ", salary=" + salary

+ "]";

}

}

**Config File**

<?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 = "emp" class = "com.ashok.spring.core.ioc.beanfactory.beans.Employee">

<property name = "empName" value = "Ashok Kumar"/>

<property name = "empId" value = "Emp0087"/>

<property name = "empAddress" value = "Bhimavaram"/>

<property name = "salary" value = "47000"/>

</bean>

</beans>

**Client Application**

package com.ashok.spring.core.ioc.beanfactory.test;

import org.springframework.beans.factory.BeanFactory;

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

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import com.ashok.spring.core.ioc.beanfactory.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

@SuppressWarnings("deprecation")

public class TestSpringApplication {

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/ioc/beanfactory/config/beans.xml";

Resource resource = new ClassPathResource(configFile);

BeanFactory factory = new XmlBeanFactory(resource);

Employee emp = (Employee) factory.getBean("emp");

System.out.println(emp);

}

}

In the above application, when we activate the BeanFactory container, then BeanFactory Container will be started. It will not create any Bean object immediately, and it will perform the following actions.

1. It will take the bean configuration file name and location from the Resource object.
2. It will search for the respective bean configuration file at the specified location.
3. If the respective bean configuration file is available, then the BeanFactory container will load that XML file to the memory.
4. After XML file loading, BeanFactory Container will parse that XML file, i.e., it will check all the tags in the XML file are provided properly or not, all attributes are available properly or not.
5. After the XML file parsing, BeanFactory Container will read data from the Beans configuration file and store it in the Resource object.

After getting BeanFactory Object, when we access the getBean(-) method, then BeanFactory will perform the following actions.

1. BeanFactory will search for the respective Bean configuration in the Resource object based on the provided identity.
2. If any bean configuration is identified in the Resource object based on the provided identity, the BeanFactory container will take the respective bean class name and location.
3. BeanFactory Container will search for the respective bean class at the specified location. If it is available, then BeanFactory Container will load all the bean class bytecode to the memory.
4. BeanFactory Container will create an Object for the loaded bean class and its dependent bean objects.
5. BeanFactory Container will store the generated bean object, and its dependent objects in the Container object in the form of Key-Value pairs, where keys must be the “id” attribute values specified in the beans configuration file and values are Bean Objects.

###### ****2. ApplicationContext****

ApplicationContext IOC Container is an extension of BeanFactory IOC Container. It can provide some advanced features like Internationalization, Event Handling etc., along with fundamental functionalities that BeanFactory provides. In Spring, ApplicationContext IOC Container is represented in the form of the following predefined interface.

"org.springframework.context.ApplicationContext".

Spring Framework has provided ApplicationContext as a child interface to the BeanFactory interface. Spring Framework has provided the following three implementation classes for ApplicationContext.

###### ****1. ClassPathXmlApplicationContext****

It can get all the beans configuration details from the configuration file in the application classpath.

###### ****2. FileSystemXmlApplicationContext****

It can get all the beans configuration details from the Configuration file, which is existed on our system hard disk.

###### ****3. WebXmlApplicationContext****

It can get all the beans configuration details from the configuration file in the web application.

E.g

**Bean Class**

package com.ashok.spring.core.ioc.applicationcontext.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

private String empName;

private String empId;

private String empAddress;

private double salary;

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getEmpAddress() {

return empAddress;

}

public void setEmpAddress(String empAddress) {

this.empAddress = empAddress;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Employee [empName=" + empName + ", empId=" + empId + ", empAddress=" + empAddress + ", salary=" + salary

+ "]";

}

}

**Config File (applicationContext.xml)**

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

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

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

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

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

<bean id = "emp" class = "com.ashok.spring.core.ioc.applicationcontext.beans.Employee">

<property name = "empName" value = "Ashok Kumar"/>

<property name = "empId" value = "Emp0087"/>

<property name = "empAddress" value = "Bhimavaram"/>

<property name = "salary" value = "50000"/>

</bean>

</beans>

**Client Application**

package com.ashok.spring.core.ioc.applicationcontext.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.ioc.applicationcontext.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

@SuppressWarnings("resource")

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/ioc/applicationcontext/config/applicationContext.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp);

}

}

When we activate the ApplicationContext container in the above application, then ApplicationContext Container will perform the following actions.

1. It will take the bean configuration file name and location from the Container class constructor.
2. It will search for the respective bean configuration file at the specified location.
3. If the respective bean configuration file is available, the ApplicationContext container will load that XML file to the memory.
4. After XML file loading, ApplicationContext Container will parse that XML file, i.e., it will check all the tags in the XML file are provided properly or not, all attributes are available properly or not.
5. After the XML file parsing, ApplicationContext Container will read data from the Beans configuration file.
6. If any bean configuration is identified in the beans configuration file, the ApplicationContext container will take beans classes and their locations.
7. ApplicationContext Container will search for the respective beans at the specified locations. If they are available, then IOC Container will load all the bean classes bytecode to the memory.
8. ApplicationContext Container will create Objects for the loaded bean classes and their dependent bean objects.
9. ApplicationContext Container will store all the bean objects and their dependent objects in the Container object in the form of Key-Value pairs, where keys must be the “id” attribute values specified in the beans configuration file.

In the above context, if we access the getBean(“–“) method over Container reference, then ApplicationContext Container will search for the Bean object based on the provided bean identity. If it is available, then AppliationContext Container will return the Bean object.

###### ****Differences between BeanFactory and ApplicationContext IOC Containers****

1. BeanFactory is a fundamental IOC Container. It can provide fundamental functionalities to the spring applications like creating and maintaining bean objects.  
   ApplicationContext IOC Container is an extension of BeanFactory IOC Container. It can provide some advanced features like Internationalization, Event Handling etc., along with fundamental functionalities that BeanFactory provides.
2. BeanFactory is not supporting integrating AOP services like Security, JTA etc., to the spring applications.  
   ApplicationContext is supporting to integrating AOP services like Security, JTA etc., into the spring applications.
3. BeanFactory is not suitable for web applications that we will prepare based on the Spring web module.  
   ApplicationContext is suitable for the web applications which we want to prepare based on the Spring web module.
4. BeanFactory can prepare Singleton objects when we send the first request for bean, Lazy Instantiation/Initialization.  
   ApplicationContext can prepare Singleton objects when we activate Container, that is, early Instantiation/Initialization.
5. BeanFactory supports only the scopes like Singleton and Prototype.  
   ApplicationContext supports almost all the Spring scopes like Singleton, Prototype, request, session, global session, WebSocket etc.
6. BeanFactory is mainly for Standalone Applications.  
   ApplicationContext is for all types of Spring framework applications.
7. BeanFactory is an outdated Container in Spring applications.  
   ApplicationContext is not an outdated Container.

**Bean Definition**

In this tutorial, we will discuss bean definition in the spring framework. Bean is a Software Reusable Component. It is a regular java class containing properties and the corresponding setXXX(-) and getXXX() methods created and managed by IOC Container in Spring Framework.

**Rules and Regulations to write Bean classes**

* Bean classes must be POJO classes, and they must not extend or implement any predefined Library except java.io.Serializable marker interface.
* Bean must be declared as “public”, “Non-abstract,” and “non-final.”

The main intention of declaring bean class as **“public”** is to make bean class scope available to IOC Container to create objects. The main purpose of declaring the bean class as **“Non-abstract”** is to create an object. The main intention to declare bean classes as**“Non-final”** is to extend one bean class to another bean class to improve reusability.

* In Bean classes, we have to declare all properties as “private” and all behaviors as “public,” which will improve “Encapsulation.”
* If we want to provide any constructor in the bean class, then provide a constructor. It must be a 0-arg constructor and “public” constructor because IOC Container will search and execute public and 0-arg constructor while instantiating bean.

Suppose we want to use Beans in Spring applications. In that case, we must configure that bean classes in the spring beans configuration file because IOC Container will recognize and create Bean objects by getting bean class details from the beans configuration file only. There are three ways to provide beans configurations in spring applications.

1. XML Configuration
2. Java Based Configuration
3. Annotations Configuration

**1. XML Configuration**

To provide beans configurations in the beans configuration file, we have to use the following XML tags.

<beans>

<bean id="--" name="--" class="--" scope="--">

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

</bean>

</beans>

Where,

* The “<beans>” tag is the root tag in the beans configuration file.
* The “<bean>” tag can provide configuration details of a particular bean class.
* The “class” attribute in can provide a fully qualified bean class name.
* The “<property> tag can represent a particular property [variable] in bean class, and it will set the specified value to the respective bean property by executing setXXX(-) method.

**Difference between “id” attribute and “name” attribute in <bean> tag?**

* **‘id’** attribute can take exactly one identity to the bean object, and it will not allow more than one identity.
* **‘name’** attribute in <bean> tag can allow more than one identity name to the bean object, wherein multiple values only first value is treated as the actual bean identity. The remaining names are alias names for the bean object. In this context, while providing alias names to the bean object, we have to use either ‘,’ or ‘;’ or [space] as delimiter[separator].

E.g 1

<beans>

<bean id="emp1" class="com.ashok.spring.beans.Employee"/>

</beans>

Employee emp = (Employee)context.getBean("emp1");

Output: Valid.

E.g 2

<beans>

<bean id="emp1 emp2 emp3" class="com.ashok.spring.beans.Employee"/>

</beans>

Employee emp = (Employee)context.getBean("emp1");

Output: org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named ' emp1 ' is defined.

**Note**

Similarly, the following cases are also be invalid.

<bean id="emp1,emp2,emp3" class=" com.ashok.spring.beans.Employee "/>->Invalid

<bean id="emp1; emp2; emp3" class=" com.ashok.spring.beans.Employee "/>->Invalid

<bean id="emp1emp2emp3" class=" com.ashok.spring.beans.Employee "/>-> Valid

E.g 3

<beans>

<bean name = "emp1" class = "com.ashok.spring.beans.Employee"/>

</beans>

Employee emp = (Employee) context.getBean(“emp1”);

**Output: Valid.**

E.g 4

<beans>

<bean name="emp1 emp2 emp3" class="com.ashok.spring.beans.Employee"/>

</beans>

Employee emp = (Employee)context.getBean(“emp1”);

**Output: Valid**

**Note**

Similarly, the following cases are also be valid.

<bean name="emp1,emp2,emp3" class=" com.ashok.spring.beans.Employee "/>

<bean name="emp1; emp2; emp3" class=" com.ashok.spring.beans.Employee "/>

**Note**

It is possible to use both ‘id’ attribute and ‘name’ attribute in single <bean> tag.

E.g 5

<beans>

<bean id=”emp1” name="emp2" class="com.ashok.spring.beans.Employee"/>

</beans>

Employee emp1 = (Employee)context.getBean(“emp1”);  
Employee emp2 = (Employee)context.getBean(“emp2”);

**Note**

It is possible to provide bean alias names explicitly from outside of the bean definition in configuration file by using <alias> tag.

<alias name="--" alias="--"/>

Where “name” attribute will take bean logical name which we specified with “id” attribute in beans configuration file. Where “alias” attribute will take alias name.

<beans>

<bean name="emp1" class=" com.ashok.spring.beans.Employee "/>

<alias name="emp1" alias="emp2"/>

<alias name="emp2" alias="emp3"/>

</bean>

</beans>

Employee emp1 = (Employee)context.getBean(“emp1”); // Valid  
Employee emp2 = (Employee)context.getBean(“emp2”); // Valid  
Employee emp3 = (Employee)context.getBean(“emp3”); // Valid

In J2SE applications, we can define scopes to the data by using the access modifiers like public, protected, , and private. Similarly, in the Spring framework, to define scopes to the beans spring framework has provided the following scopes.

1. Singleton Scope [Default Scope]
2. Prototype Scope
3. Request Scope
4. Session Scope
5. GlobalSession Scope
6. Application Scope
7. WebSocket scope

**1. Singleton Scope**

* It is the default scope in Spring applications.
* If we use this scope to the bean, IOC Container will create a Single Bean object for the single bean definition in the Spring config file.
* This approach will return the same bean object every time requesting the bean object.
* When we request a bean object the first time, IOCContainer will create a bean object and store it in Cache memory. Then, every time accessing the bean object, IOCContainer will return the same bean object reference value without creating new Bean objects.

E.g

<bean id = "emp" class = "com.ashok.spring.core.bean.scopes.singleton.beans.Employee" scope="singleton">

System.out.println(context.getBean("emp"));// Employee @a111

System.out.println(context.getBean("emp"));// Employee @a111

**2. Prototype Scope**

* It is not the default Scope in the Spring framework.
* In Spring applications, if we provide a “prototype” scope in the bean configuration file, then IOC Container will create a new Bean object when calling the getBean(–) method.

E.g.

<bean id = "emp" class = "com.ashok.spring.core.bean.scopes. prototype.beans.Employee" scope="prototype">

System.out.println(context.getBean("emp"));// Employee @a111

System.out.println(context.getBean("emp"));// Employee @a222

System.out.println(context.getBean("emp"));// Employee @a333

**3. Request Scope**

* This scope is not helpful in Standalone Applications [Spring Core Module]. It will be used in Web applications that are prepared based on the Spring Web module.
* Request Scope can create a separate bean object for every request object.

**4. Session Scope**

* Session Scope is used in web applications that are prepared based on the Spring web module, and it is not applicable in Standalone Applications.
* Session Scope allows creating a separate bean object for every Session object in web applications.

**5. GlobalSession Scope**

* This scope is not helpful in standard applications, and it is useful in portlet applications which are prepared based on the Spring web module.
* Global Session scope allows creating a separate bean object for every portlet Session.

**6. Application Scope**

* This scope is not helpful in standalone Applications, and it is useful in web applications prepared based on the Spring web module.
* ApplicationScope allows the creation of a separate bean object for every ServletContext object.

**7. Websocket Scope**

* This scope is useful in web applications that are prepared based on the spring web module.
* WebSocket scope allows the creation of a separate bean object for a single WebSocket lifecycle.
* Suppose we use the scopes like request, session, globalSession, application, WebSocket, etc., in standalone applications which are prepared based on the spring core module. In that case, Container will raise an exception like “java.lang.IllegalStateException”.

**Note**

Spring Framework has provided an environment to customize the existed scopes, but it is not suggestible. The spring framework has provided an environment to create new scopes in spring applications.

**2. Java Based Configuration**

In Spring, up to Spring2.4 version, the Spring beans configuration file is mandatory to configure bean classes and their metadata. Right from the Spring3.x version, the Spring beans configuration file is optional because the Spring3.x version has provided Java Based Configuration as a replacement for XML documents.

If we want to use Java Based Configuration as an alternative to the Spring beans configuration file in Spring applications, we must use the following steps.

1. Create Bean classes as per the requirement.  
2. Create Beans configuration class with the following annotations.

**org.springframework.context.annotation.@Configuration:** It able to represent a class as configuration class.  
**org.springframework.context.annotation.@Bean:** It will be used at method to represent the return object is bean object.

3. In Test class, Create ApplicationContext object with the

org.springframework.context.annotation.AnnotationConfigApplicationContext implementation class.

ApplicationContext context = new AnnotationConfigApplicationContext(BeanConfig.class);

4. Get Bean object from ApplicationContext by using the following method.

public Object getaBean(Class c)

E.g.

Employee emp = context.getBean(Employee.class);

5. Access business methods from Bean.

E.g

**Bean class**

package com.ashok.spring.core.bean.javabasedconfig.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

private String empName;

private String empId;

private String empAddress;

private double salary;

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getEmpAddress() {

return empAddress;

}

public void setEmpAddress(String empAddress) {

this.empAddress = empAddress;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Employee [empName=" + empName + ", empId=" + empId + ", empAddress=" + empAddress + ", salary=" + salary

+ "]";

}

}

**EmployeeConfig.java**

package com.ashok.spring.core.bean.javabasedconfig.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.ashok.spring.core.bean.javabasedconfig.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

@Configuration

public class EmployeeConfig {

@Bean

public Employee getEmployeeConfig() {

return new Employee();

}

}

**Client Application**

package com.ashok.spring.core.bean.javabasedconfig.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.ashok.spring.core.bean.javabasedconfig.beans.Employee;

import com.ashok.spring.core.bean.javabasedconfig.config.EmployeeConfig;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

@SuppressWarnings("resource")

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(EmployeeConfig.class);

Employee emp = (Employee) context.getBean("getEmployeeConfig");

emp.setEmpName("Ashok Kumar");

emp.setEmpAddress("Bhimavaram");

emp.setEmpId("E0087");

emp.setSalary(45000);

System.out.println(emp);

}

}

**Post navigation**

Prev

**Bean Life Cycle**

In this tutorial, we are going to discuss the bean life cycle in the Spring framework. In spring framework applications, when IOC Container recognizes all the beans definitions in the beans configuration file, IOC Container will execute that bean using the following lifecycle actions.

1. Bean Class Loading
2. Bean Instantiation
3. Bean Initialization and Destruction



**1. Bean Class Loading**

When IOC Container recognizes fully qualified names of the bean classes in the beans configuration file, IOC Container will load the specified bean class byte code to the memory. To load bean class byte code to the memory, IOC Container will use the following method.

public static Class forName(String class\_Name)throws ClassNotFoundException

E.g.

Class c = Class.forName(“com.ashok.spring.core.bean.scopes.prototype.beans.Employee");

**2. Bean Instantiation**

In Spring applications, after loading bean class byte code to the memory, IOC Container will create an object for the bean class. In Spring applications, we can use the following three approaches to create Bean objects.

1. By using Constructor directly.
2. By Using Static Factory Method
3. By Using Instance Factory Method

**1. Bean Instantiation through Constructors**

If we want to create bean objects using constructors, we must provide a 0-arg constructor in the bean class irrespective of bean class constructor scopes.

**Note**

In Bean class, if we provide a parameterized constructor, then IOC Container will raise an exception.

E.g

**Bean Class**

package com.ashok.spring.core.bean.instantiation.constructor.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

public Employee() {

System.out.println("Bean Instantiation using constructor");

}

public String getStatus() {

return "Hey..!! Bean created using constructor..!!";

}

}

**Config File**

<?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="emp"

class="com.ashok.spring.core.bean.instantiation.constructor.beans.Employee">

</bean>

</beans>

**Client Application**

package com.ashok.spring.core.bean.instantiation.constructor.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.instantiation.constructor.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

@SuppressWarnings("resource")

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/instantiation/constructor/config/applicationContext.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp.getStatus());

}

}

Output

Bean Instantiation using constructor

Hey...!! Bean created using constructor...!!

**2. Bean Instantiation through Static Factory Method**

In this approach, first, we have to define the static factory method in the Bean class. We have to configure that static factory method in the bean definition in the beans configuration file.

E.g.

**Bean Class**

package com.ashok.spring.core.bean.instantiation.staticfactory.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

public static Employee getInstance() {

System.out.println("Bean Instantiation using static factory..!!");

return new Employee();

}

public String getStatus() {

return "Hey..!! Bean created using static factory..!!";

}

}

**Config File**

<?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="emp"

class="com.ashok.spring.core.bean.instantiation.staticfactory.beans.Employee"

factory-method="getInstance">

</bean>

</beans>

**Client Application**

package com.ashok.spring.core.bean.instantiation.staticfactory.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.instantiation.staticfactory.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

@SuppressWarnings("resource")

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/instantiation/staticfactory/config/applicationContext.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp.getStatus());

}

}

Output

Bean Instantiation using static factory...!!

Hey...!! Bean created using static factory...!!

**3. Bean Instantiation through Instance Factory Method**

In this approach, we have to define a separate factory class with an instance factory method, and we have to configure the factory class as a bean in the beans configuration file. Then we have to configure the factory class and factory method in the original bean class definition by using “factory-bean” and “factory-method” attributes.

E.g

**Bean Class**

package com.ashok.spring.core.bean.instantiation.instancefactory.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

public String getStatus() {

return "Hey..!! Bean created using instance factory..!!";

}

}

**Factory Class**

package com.ashok.spring.core.bean.instantiation.instancefactory.factory;

import com.ashok.spring.core.bean.instantiation.instancefactory.beans.Employee;

/\*\*

\*

\* @author ashok.mariyala

\*

\*/

public class EmployeeFactory {

public Employee getInstance() {

System.out.println("Bean Instantiation using instance factory..!!");

return new Employee();

}

}

**Config File**

<?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="emp"

class="com.ashok.spring.core.bean.instantiation.instancefactory.beans.Employee"

factory-method="getInstance" factory-bean="factory">

</bean>

<bean id="factory" class="com.ashok.spring.core.bean.instantiation.instancefactory.factory.EmployeeFactory" />

</beans>

**Client Application**

package com.ashok.spring.core.bean.instantiation.instancefactory.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.instantiation.instancefactory.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

@SuppressWarnings("resource")

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/instantiation/instancefactory/config/applicationContext.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp.getStatus());

}

}

Output

Bean Instantiation using instance factory...!!

Hey...!! Bean created using instance factory...!!

**3. Bean Initialization and Bean Destruction**

As part of the Beans lifecycle, IOC Container has to perform Beans initialization after the Bean Instantiation. IOC Container has to perform Bean destruction after executing the business logic or at the time of shutdown the IOC Container. There are three ways to perform Beans initialization and destruction in Spring Framework.

1. By using Custom initialization and destruction methods.
2. By using InitializingBean and DesposableBean callback interfaces.
3. By using @PostConstruct and @Predestroy annotations

**1. By using Custom initialization and destruction methods**

In this approach, we have to define user-defined initialization and destruction methods with any name, and we have to configure those user-defined methods in beans definitions in the beans configuration file by using the “init-method” and “destroy-method” attribute in the tag.

E.g

package com.ashok.spring.core.bean.initialization.custom.beans;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

public void init() {

System.out.println("User defined init() method");

}

public void destroy() {

System.out.println("User defined destroy() method");

}

public void getMessage() {

System.out.println("Inside Employee class..!!");

}

}

<?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="emp"

class="com.ashok.spring.core.bean.initialization.custom.beans.Employee"

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

</beans>

package com.ashok.spring.core.bean.initialization.custom.test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.initialization.custom.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/initialization/custom/config/applicationContext.xml";

AbstractApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp);

context.close();

}

}

Output

User defined init() method

com.ashok.spring.core.bean.initialization.custom.beans.Employee@3e9b1010

User defined destroy() method

**2. By using InitializingBean and DesposableBean callback interfaces**

In the Spring framework, InitializingBean is a callback interface, and it provides the following method to execute while performing bean initialization by IOC Container.

public void afterPropertiesSet()throws Exception

**Note**

The IOC container will execute this method after executing all the setXXX() bean class methods by ApplicationContext.

In the Spring framework, DisposableBean is a callback interface, and it provides the following method to execute while performing Bean Destruction by IOC Container.

public void destroy()

E.g

package com.ashok.spring.core.bean.initialization.callbackinterface.beans;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee implements InitializingBean, DisposableBean {

@Override

public void destroy() {

System.out.println("Inside destroy() method");

}

public void getMessage() {

System.out.println("Inside Employee class..!!");

}

@Override

public void afterPropertiesSet() throws Exception {

System.out.println("Inside afterPropertiesSet() method");

}

}

<?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="emp"

class="com.ashok.spring.core.bean.initialization.callbackinterface.beans.Employee" />

</beans>

package com.ashok.spring.core.bean.initialization.callbackinterface.test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.initialization.callbackinterface.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/initialization/callbackinterface/config/applicationContext.xml";

AbstractApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp);

context.close();

}

}

Output

Inside afterPropertiesSet() method

com.ashok.spring.core.bean.initialization.callbackinterface.beans.Employee@3e9b1010

Inside destroy() method

**3. By using @PostConstruct and @Predestroy annotations**

* @PostConstruct annotation will make a method to execute by the IOC Container while performing Beans initialization.
* @Predestroy annotation will make a method to execute by the IOC Container while performing Bean Destruction.

package com.ashok.spring.core.bean.initialization.annotations.beans;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class Employee {

@PreDestroy

public void destroy() {

System.out.println("Inside @@PreDestroy annotation method");

}

public void getMessage() {

System.out.println("Inside Employee class..!!");

}

@PostConstruct

public void init() throws Exception {

System.out.println("Inside @PostConstruct annotation method");

}

}

<?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="emp"

class="com.ashok.spring.core.bean.initialization.annotations.beans.Employee" />

</beans>

package com.ashok.spring.core.bean.initialization.annotations.test;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.ashok.spring.core.bean.initialization.annotations.beans.Employee;

/\*\*

\*

\* @author Ashok Kumar

\*

\*/

public class TestSpringApplication {

public static void main(String[] args) {

String configFile = "/com/ashok/spring/core/bean/initialization/annotations/config/applicationContext.xml";

AbstractApplicationContext context = new ClassPathXmlApplicationContext(configFile);

Employee emp = (Employee) context.getBean("emp");

System.out.println(emp);

context.close();

}

}

Inside @PostConstruct annotation method

com.ashok.spring.core.bean.initialization.annotations.beans.Employee@57f23557

Inside @@PreDestroy annotation method

**Note**

In general, in spring framework applications, we can use either of the above three approaches to performing beans initialization and destruction. We are not using all three approaches at a time. Suppose we use all the above three approaches in a single bean. In that case, IOC Container will execute the above three approaches provided initialization methods and destruction methods in the following order.

**Initialization order**

1. an initialization method marked with @Postconstruct annotation.
2. afterPropertiesSet() method provided by InnitializingBean callback interface.
3. an initialization method configured with “init-method” in <bean> tag in beans configuration file

**Destruction Order**

1. A Destruction method marked with @Predestroy annotation
2. destroy() method provided by DisposableBean callback interface.
3. A destruction method configured with “destroy-method” annotation in <bean> tag in beans configuration file.