Spring:

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as [Struts](https://www.javatpoint.com/struts-2-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), Tapestry, [EJB](https://www.javatpoint.com/ejb-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Servlets->Spring🡪springmvc-🡪spring boot with microservices

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as [JMS](https://www.javatpoint.com/jms-tutorial)

, [JDBC](https://www.javatpoint.com/java-jdbc)

, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

IOC(Inversion of Control):

# **IoC Container**

1. [IoC Container](https://www.javatpoint.com/ioc-container)
2. [Using BeanFactory](https://www.javatpoint.com/ioc-container)
3. [Using ApplicationContext](https://www.javatpoint.com/ioc-container)

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations 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**

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

32.1M

642

HTML Tutorial

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

1. ApplicationContext context =
2. **new** ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

# **Dependency Injection by Constructor Example**

1. [Dependency Injection by constructor](https://www.javatpoint.com/spring-tutorial-dependency-injection-by-constructor)
2. [Injecting primitive and string-based values](https://www.javatpoint.com/spring-tutorial-dependency-injection-by-constructor#ips)

We can inject the dependency by constructor. The **<constructor-arg>** subelement of **<bean>** is used for constructor injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

### Injecting primitive and string-based values

Let's see the simple example to inject primitive and string-based values. We have created three files here:

* Employee.java
* applicationContext.xml
* Test.java

**Employee.java**

It is a simple class containing two fields id and name. There are four constructors and one method in this class.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. **public** Employee() {System.out.println("def cons");}
9. **public** Employee(**int** id) {**this**.id = id;}
11. **public** Employee(String name) {  **this**.name = name;}
13. **public** Employee(**int** id, String name) {
14. **this**.id = id;
15. **this**.name = name;
16. }
18. **void** show(){
19. System.out.println(id+" "+name);
20. }
22. }

**applicationContext.xml**

We are providing the information into the bean by this file. The constructor-arg element invokes the constructor. In such case, parameterized constructor of int type will be invoked. The value attribute of constructor-arg element will assign the specified value. The type attribute specifies that int parameter constructor will be invoked.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="e" **class**="com.javatpoint.Employee">
10. <constructor-arg value="10" type="int"></constructor-arg>
11. </bean>
13. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the show method.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.\*;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Employee s=(Employee)factory.getBean("e");
14. s.show();
16. }
17. }

**Output:**10 null

[download this example](https://static.javatpoint.com/src/sp/dicons.zip)

### Injecting string-based values

If you don't specify the type attribute in the constructor-arg element, by default string type constructor will be invoked.

1. ....
2. <bean id="e" **class**="com.javatpoint.Employee">
3. <constructor-arg value="10"></constructor-arg>
4. </bean>
5. ....

If you change the bean element as given above, string parameter constructor will be invoked and the output will be 0 10.

**Output:**0 10

You may also pass the string literal as following:

1. ....
2. <bean id="e" **class**="com.javatpoint.Employee">
3. <constructor-arg value="Sonoo"></constructor-arg>
4. </bean>
5. ....

**Output:**0 Sonoo

You may pass integer literal and string both as following

1. ....
2. <bean id="e" **class**="com.javatpoint.Employee">
3. <constructor-arg value="10" type="int" ></constructor-arg>
4. <constructor-arg value="Sonoo"></constructor-arg>
5. </bean>
6. ....

# **Constructor Injection with Dependent Object**

1. [Constructor Injection with Dependent Object](https://www.javatpoint.com/spring-tutorial-constructor-injection-with-dependent-object)

If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object. Let's see the Address class first:

**Address.java**

This class contains three properties, one constructor and toString() method to return the values of these object.

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String city;
5. **private** String state;
6. **private** String country;
8. **public** Address(String city, String state, String country) {
9. **super**();
10. **this**.city = city;
11. **this**.state = state;
12. **this**.country = country;
13. }
15. **public** String toString(){
16. **return** city+" "+state+" "+country;
17. }
18. }

**Employee.java**

It contains three properties id, name and address(dependent object) ,two constructors and show() method to show the records of the current object including the depedent object.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** Address address;//Aggregation
8. **public** Employee() {System.out.println("def cons");}
10. **public** Employee(**int** id, String name, Address address) {
11. **super**();
12. **this**.id = id;
13. **this**.name = name;
14. **this**.address = address;
15. }
17. **void** show(){
18. System.out.println(id+" "+name);
19. System.out.println(address.toString());
20. }
22. }

**applicationContext.xml**

The **ref** attribute is used to define the reference of another object, such way we are passing the dependent object as an constructor argument.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="a1" **class**="com.javatpoint.Address">
10. <constructor-arg value="ghaziabad"></constructor-arg>
11. <constructor-arg value="UP"></constructor-arg>
12. <constructor-arg value="India"></constructor-arg>
13. </bean>

16. <bean id="e" **class**="com.javatpoint.Employee">
17. <constructor-arg value="12" type="int"></constructor-arg>
18. <constructor-arg value="Sonoo"></constructor-arg>
19. <constructor-arg>
20. <ref bean="a1"/>
21. </constructor-arg>
22. </bean>
24. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the show method.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.\*;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Employee s=(Employee)factory.getBean("e");
14. s.show();
16. }
17. }

# **Dependency Injection by setter method**

1. [Dependency Injection by constructor](https://www.javatpoint.com/spring-tutorial-dependency-injection-by-setter-method)
2. [Injecting primitive and string-based values](https://www.javatpoint.com/spring-tutorial-dependency-injection-by-setter-method#ips)

We can inject the dependency by setter method also. The **<property>** subelement of **<bean>** is used for setter injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

### Injecting primitive and string-based values by setter method

Let's see the simple example to inject primitive and string-based values by setter method. We have created three files here:

* Employee.java
* applicationContext.xml
* Test.java

**Employee.java**

It is a simple class containing three fields id, name and city with its setters and getters and a method to display these informations.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** String city;
8. **public** **int** getId() {
9. **return** id;
10. }
11. **public** **void** setId(**int** id) {
12. **this**.id = id;
13. }
14. **public** String getName() {
15. **return** name;
16. }
17. **public** **void** setName(String name) {
18. **this**.name = name;
19. }
21. **public** String getCity() {
22. **return** city;
23. }
24. **public** **void** setCity(String city) {
25. **this**.city = city;
26. }
27. **void** display(){
28. System.out.println(id+" "+name+" "+city);
29. }
31. }

**applicationContext.xml**

We are providing the information into the bean by this file. The property element invokes the setter method. The value subelement of property will assign the specified value.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="obj" **class**="com.javatpoint.Employee">
10. <property name="id">
11. <value>20</value>
12. </property>
13. <property name="name">
14. <value>Arun</value>
15. </property>
16. <property name="city">
17. <value>ghaziabad</value>
18. </property>
20. </bean>
22. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display method.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.\*;
7. **public** **class** Test {
8. **public** **static** **void** main(String[] args) {
10. Resource r=**new** ClassPathResource("applicationContext.xml");
11. BeanFactory factory=**new** XmlBeanFactory(r);
13. Employee e=(Employee)factory.getBean("obj");
14. s.display();
16. }
17. }

# **Setter Injection with Dependent Object Example**

1. [Setter Injection with Dependent Object](https://www.javatpoint.com/spring-tutorial-setter-injection-with-dependent-object)

Like Constructor Injection, we can inject the dependency of another bean using setters. In such case, we use **property** element. Here, our scenario is **Employee HAS-A Address**. The Address class object will be termed as the dependent object. Let's see the Address class first:

**Address.java**

This class contains four properties, setters and getters and toString() method.

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String addressLine1,city,state,country;
6. //getters and setters
8. **public** String toString(){
9. **return** addressLine1+" "+city+" "+state+" "+country;
10. }

**Employee.java**

It contains three properties id, name and address(dependent object) , setters and getters with displayInfo() method.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** Address address;
8. //setters and getters
10. **void** displayInfo(){
11. System.out.println(id+" "+name);
12. System.out.println(address);
13. }
14. }

**applicationContext.xml**

The **ref** attribute of **property** elements is used to define the reference of another bean.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="address1" **class**="com.javatpoint.Address">
10. <property name="addressLine1" value="51,Lohianagar"></property>
11. <property name="city" value="Ghaziabad"></property>
12. <property name="state" value="UP"></property>
13. <property name="country" value="India"></property>
14. </bean>
16. <bean id="obj" **class**="com.javatpoint.Employee">
17. <property name="id" value="1"></property>
18. <property name="name" value="Sachin Yadav"></property>
19. <property name="address" ref="address1"></property>
20. </bean>
22. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.context.ApplicationContext;
6. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
7. **import** org.springframework.core.io.ClassPathResource;
8. **import** org.springframework.core.io.Resource;
10. **public** **class** Test {
11. **public** **static** **void** main(String[] args) {
12. Resource r=**new** ClassPathResource("applicationContext.xml");
13. BeanFactory factory=**new** XmlBeanFactory(r);
15. Employee e=(Employee)factory.getBean("obj");
16. e.displayInfo();
18. }
19. }