Spring Tutorial

This spring tutorial provides in-depth concepts of Spring Framework with simplified examples. It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

It is helpful for beginners and experienced persons.

Spring Framework

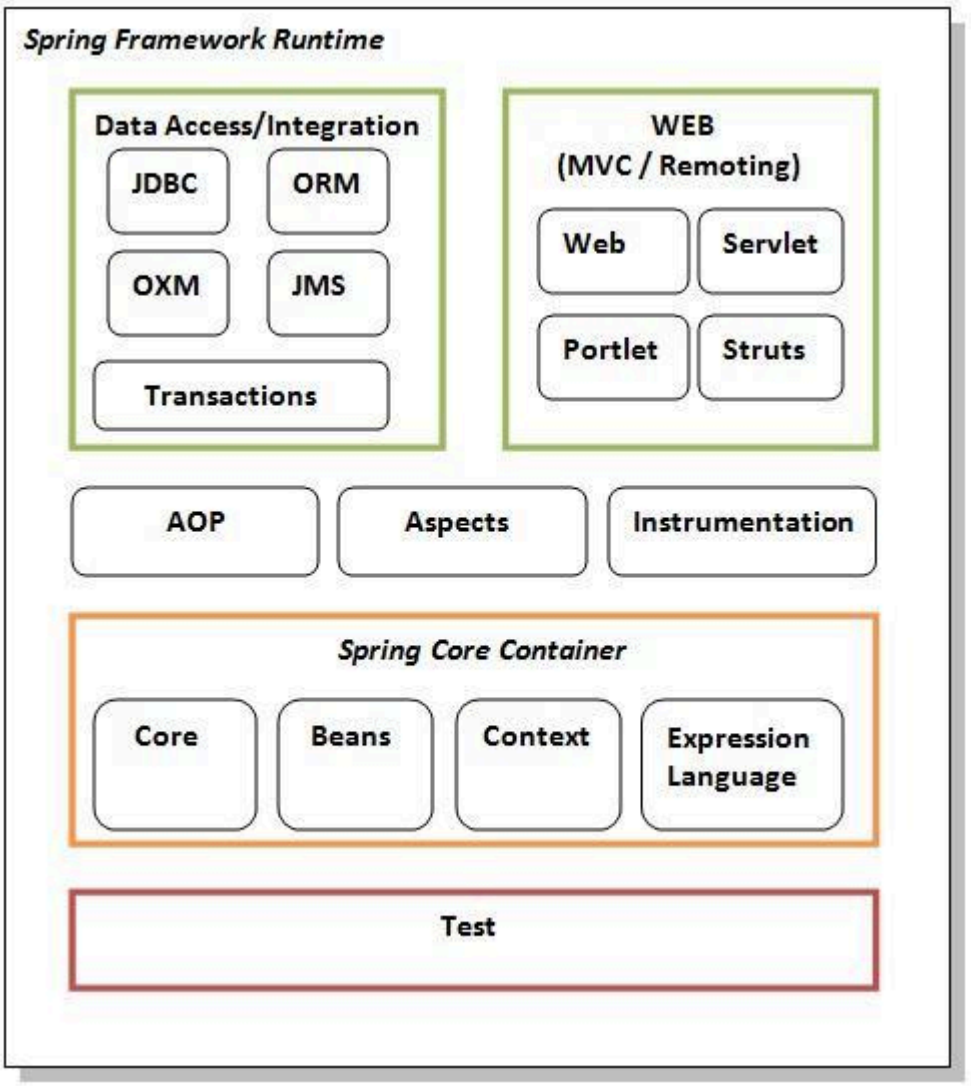
Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

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

Spring Modules

1. Spring Modules
2. Test
3. Spring Core Container
4. AOP, Aspects and Instrumentation
5. Data Access / Integration
6. Web

The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc. These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, Web (MVC / Remoting) as displayed in the following diagram.



Test

This layer provides support of testing with JUnit and TestNG.

Spring Core Container

The Spring Core container contains core, beans, context and expression language (EL) modules.

Core and Beans

These modules provide IOC and Dependency Injection features.

Context

This module supports internationalization (I18N), EJB, JMS, Basic Remoting.

Expression Language

It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

AOP, Aspects and Instrumentation

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

The aspects module provides support to integration with AspectJ.

The instrumentation module provides support to class instrumentation and classloader implementations.

Data Access / Integration

This group comprises of JDBC, ORM, OXM, JMS and Transaction modules. These modules basically provide support to interact with the database.

Web

This group comprises of Web, Web-Servlet, Web-Struts and Web-Portlet. These modules provide support to create web application.

Spring Example

1. Steps to create spring application

Here, we are going to learn the simple steps to create the first spring application. To run this application, we are not using any IDE. We are simply using the command prompt. Let's see the simple steps to create the spring application

- **create the class**
- **create the xml file to provide the values**
- **create the test class**
- **Load the spring jar files**
- **Run the test class**

Steps to create spring application

Let's see the 5 steps to create the first spring application.

1) Create Java class

This is the simple java bean class containing the name property only.

```
1.  package com.javatpoint;
2.
3.  public class Student {
4.  private String name;
5.
6.  public String getName() {
7.     return name;
8.  }
9.
10. public void setName(String name) {
11.    this.name = name;
12. }
13.
14. public void displayInfo(){
15.    System.out.println("Hello: "+name);
16. }
17. }
```

This is simple bean class, containing only one property name with its getters and setters method. This class contains one extra method named displayInfo() that prints the student name by the hello message.

2) Create the xml file

In case of myeclipse IDE, you don't need to create the xml file as myeclipse does this for yourselves. Open the applicationContext.xml file, and write the following code:

```
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">
8.
9.  <bean id="studentbean" class="com.javapoint.Student">
10. <property name="name" value="Vimal Jaiswal"></property>
11. </bean>
12.
13. </beans>
```

The **bean** element is used to define the bean for the given class. The **property** subelement of bean specifies the property of the Student class named name. The value specified in the property element will be set in the Student class object by the IOC container.

3) Create the test class

Create the java class e.g. Test. Here we are getting the object of Student class from the IOC container using the getBean() method of BeanFactory. Let's see the code of test class.

```
1.  package com.javatpoint;
2.
3.  import org.springframework.beans.factory.BeanFactory;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.ClassPathResource;
6.  import org.springframework.core.io.Resource;
7.
8.  public class Test {
9.  public static void main(String[] args) {
10.    Resource resource=new ClassPathResource("applicationContext.xml");
11.    BeanFactory factory=new XmlBeanFactory(resource);
12.
13.    Student student=(Student)factory.getBean("studentbean");
14.    student.displayInfo();
15. }
16. }
```

The **Resource** object represents the information of applicationContext.xml file. The Resource is the interface and the **ClassPathResource** is the implementation class of the Reource interface. The **BeanFactory** is responsible to return the bean. The **XmlBeanFactory** is the implementation class of the BeanFactory. There are many methods in the BeanFactory interface. One method is **getBean()**, which returns the object of the associated class.

4) Load the jar files required for spring framework

There are mainly three jar files required to run this application.

- **org.springframework.core-3.0.1.RELEASE-A**
- **com.springsource.org.apache.commons.logging-1.1.1**
- **org.springframework.beans-3.0.1.RELEASE-A**

For the future use, You can download the required jar files for spring core application.

IoC Container

1. IoC Container
2. Using BeanFactory
3. Using ApplicationContext

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:


```
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.

Dependency Injection in Spring

In such case, instance of Address class is provided by external souce 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

Upcoming topics in Spring Dependency Injection

Dependency Injection by constructor

Let's see how we can inject dependency by constructor.

Dependency Injection by Constructor Example

1. Dependency Injection by constructor
2. Injecting primitive and string-based values

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;
2.
3.  public class Employee {
4.  private int id;
5.  private String name;
6.
7.  public Employee() {System.out.println("def cons");}
8.
9.  public Employee(int id) {this.id = id;}
10.
```

```
11.  public Employee(String name) {  this.name = name;}
12.
13.  public Employee(int id, String name) {
14.     this.id = id;
15.     this.name = name;
16.  }
17.
18.  void show(){
19.     System.out.println(id+" "+name);
20.  }
21.
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">
8.
9.  <bean id="e" class="com.javapoint.Employee">
10. <constructor-arg value="10" type="int"></constructor-arg>
11. </bean>
12.
13. </beans>
```

Test.java

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

```
1.  package com.javatpoint;
2.
3.  import org.springframework.beans.factory.BeanFactory;
4.  import org.springframework.beans.factory.xml.XmlBeanFactory;
5.  import org.springframework.core.io.*;
6.
7.  public class Test {
8.     public static void main(String[] args) {
9.
10.       Resource r=new ClassPathResource("applicationContext.xml");
11.       BeanFactory factory=new XmlBeanFactory(r);
12.
13.       Employee s=(Employee)factory.getBean("e");
14.       s.show();
15.
16.    }
17. }
```

Output:10 null

download this example

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.

```
....
<bean id="e" class="com.javatpoint.Employee">
<constructor-arg value="10"></constructor-arg>
</bean>
....
```

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:

```
....
<bean id="e" class="com.javatpoint.Employee">
<constructor-arg value="Sonoo"></constructor-arg>
</bean>
....
```

Output:0 Sonoo

You may pass integer literal and string both as following

```
....
<bean id="e" class="com.javatpoint.Employee">
<constructor-arg value="10" type="int" ></constructor-arg>
<constructor-arg value="Sonoo"></constructor-arg>
</bean>
....
```

Output:10 Sonoo

Constructor Injection with Dependent Object

1. 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;
2.
3.  public class Address {
4.  private String city;
5.  private String state;
6.  private String country;
7.
8.  public Address(String city, String state, String country) {
9.     super();
10.    this.city = city;
11.    this.state = state;
12.    this.country = country;
13. }
14.
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;
2.
3.  public class Employee {
4.  private int id;
5.  private String name;
6.  private Address address;//Aggregation
7.
8.  public Employee() {System.out.println("def cons");}
9.
10. public Employee(int id, String name, Address address) {
11.    super();
12.    this.id = id;
13.    this.name = name;
14.    this.address = address;
15. }
16.
17. void show(){
18.    System.out.println(id+" "+name);
19.    System.out.println(address.toString());
20. }
21.
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.

```
<?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:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="a1" class="com.javatpoint.Address">
<constructor-arg value="ghaziabad"></constructor-arg>
<constructor-arg value="UP"></constructor-arg>
<constructor-arg value="India"></constructor-arg>
</bean>


<bean id="e" class="com.javatpoint.Employee">
<constructor-arg value="12" type="int"></constructor-arg>
<constructor-arg value="Sonoo"></constructor-arg>
<constructor-arg>
<ref bean="a1"/>
</constructor-arg>
</bean>

</beans>
```

Test.java

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

```
package com.javatpoint;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

public class Test {
   public static void main(String[] args) {

      Resource r=new ClassPathResource("applicationContext.xml");
      BeanFactory factory=new XmlBeanFactory(r);

      Employee s=(Employee)factory.getBean("e");
      s.show();

   }
}
```

Dependency Injection by setter method

1. Dependency Injection by constructor
2. Injecting primitive and string-based values

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;
2.
3.  public class Employee {
4.  private int id;
5.  private String name;
6.  private String city;
7.
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. }
20.
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. }
30.
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.


```
<?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:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="obj" class="com.javatpoint.Employee">
<property name="id" >
<value>20</value>
</property>
<property name="name">
<value>Arun</value>
</property>
<property name="city">
<value>ghaziabad</value>
</property>

</bean>

</beans>
```

Test.java

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

```
package com.javatpoint;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

public class Test {
   public static void main(String[] args) {

      Resource r=new ClassPathResource("applicationContext.xml");
      BeanFactory factory=new XmlBeanFactory(r);

      Employee e=(Employee)factory.getBean("obj");
      s.display();

   }
}
```

Output:20 Arun Ghaziabad