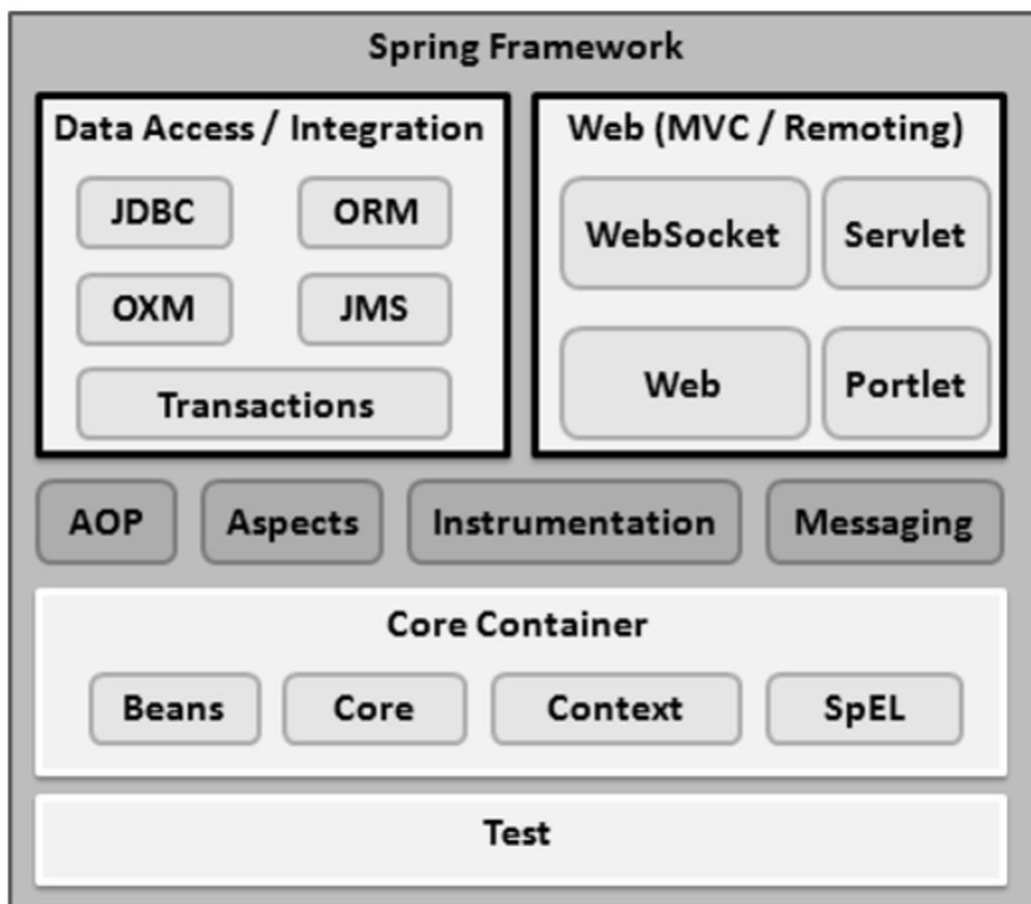


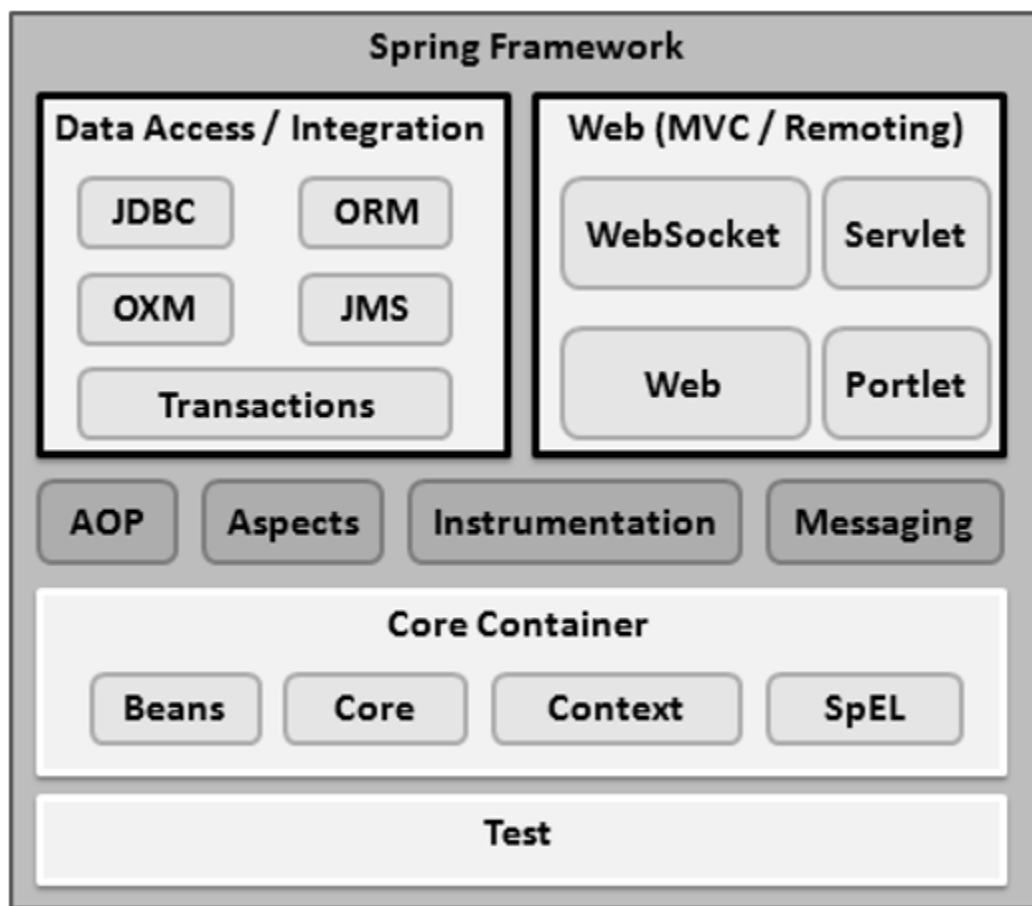
Unit 1 - Introduction to spring framework



Spring Framework - Architecture

Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows –

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the

factory pattern.

- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The `ApplicationContext` interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

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

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

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows –


- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows –

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

Useful Video Courses



Mastering Spring Framework

102 Lectures 8 hours

Karthikeya T

[More Detail](#)



Spring Boot: A Quick Tutorial Guide



Spring – Understanding Inversion of Control with Example

Last Updated : 18 Feb, 2022

Spring IoC (Inversion of Control) Container is the core of [Spring Framework](#). It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.

There are 2 types of IoC containers:

- [BeanFactory](#)
- [ApplicationContext](#)

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The following are some of the

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Got It !

- Managing our objects,
- Helping our application to be configurable,
- Managing dependencies

Implementation: So now let's understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().

Java

```
// Java Program to Illustrate Sim Interface
public interface Sim
{
    void calling();
    void data();
}
```

Now we have created another two classes Airtel and Jio which implement the Sim interface and override the interface methods.

Java

```
// Java Program to Illustrate Airtel Class

// Class
// Implementing Sim interface
public class Airtel implements Sim {

    @Override public void calling()
    {
        System.out.println("Airtel Calling");
    }

    @Override public void data()
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Got It !

Java

```
// Java Program to Illustrate Jio Class

// Class
// Implementing Sim interface
public class Jio implements Sim{
    @Override
    public void calling() {
        System.out.println("Jio Calling");
    }

    @Override
    public void data() {
        System.out.println("Jio Data");
    }
}
```

So let's now call these methods inside the main method. So by implementing the [Run time polymorphism](#) concept we can do something like this

Java

```
// Java Program to Illustrate Mobile Class

// Class
public class Mobile {

    // Main driver method
    public static void main(String[] args)
    {
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#)

Got It !

Further reading:

Wiring in Spring: @Autowired, @Resource and @Inject (/spring-annotations-resource-inject-autowire)

This article will compare and contrast the use of annotations related to dependency injection, namely the @Resource, @Inject, and @Autowired annotations.

[Read more \(/spring-annotations-resource-inject-autowire\)](/spring-annotations-resource-inject-autowire) →

@Component vs @Repository and @Service in Spring (/spring-component-repository-service)

Learn about the differences between the @Component, @Repository and @Service annotations and when to use them.

[Read more \(/spring-component-repository-service\)](/spring-component-repository-service) →

2. What Is Inversion of Control?

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

- decoupling the execution of a task from its implementation

- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

<https://www.baeldung.com/inversion-control-and-dependency-injection-i...>
We're going to look at DI next.

3. What Is Dependency Injection?

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.

Connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves.

Here's how we would create an object dependency in traditional programming:

```
public class Store {  
    private Item item;  
  
    public Store() {  
        item = new ItemImpl1();  
    }  
}
```

In the example above, we need to instantiate an implementation of the *Item* interface within the *Store* class itself.

freestar.com/?utm_campaign=branding&utm_medium=&utm_source=baeldung

By using DI, we can rewrite the example without specifying the implementation of the *Item* that we want:

```
public class Store {  
    private Item item;  
    public Store(Item item) {  
        this.item = item;  
    }  
}
```

In the next sections, we'll look at how we can provide the implementation of *Item* through metadata.

Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

4. The Spring IoC Container

An IoC container is a common characteristic of frameworks that implement IoC.

In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their life cycles.

The Spring framework provides several implementations of the *ApplicationContext* interface: *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* for standalone applications, and *WebApplicationContext* for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Here's one way to manually instantiate a container:

```
ApplicationContext context  
    = new ClassPathXmlApplicationContext("applicationContext.xml");
```

To set the *item* attribute in the example above, we can use metadata. Then the container will read this metadata and use it to assemble beans at runtime.

Dependency Injection in Spring can be done through constructors, setters or fields.

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:

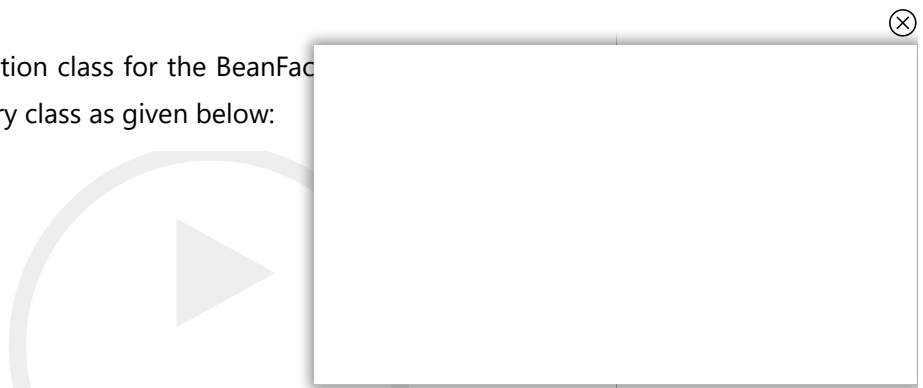
1. **BeanFactory**
2. **ApplicationContext**

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 create the instance of `XmlBeanFactory` class as given below:





```
Resource resource=new ClassPathResource("applicationContext.xml");  
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:

```
ApplicationContext context =  
    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.

[download the example to use ApplicationContext](#)

← Prev

Next →



Dependency Injection by Constructor Example

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.

```
package com.javatpoint;

public class Employee {
    private int id;
    private String name;

    public Employee() {System.out.println("def cons");}

    public Employee(int id) {this.id = id;}

    public Employee(String name) { this.name = name;}

    public void print() {
        System.out.println("id: " + id + " name: " + name);
    }
}
```

↑ SCROLL TO TOP (int id, String name) {

```
    this.id = id;
    this.name = name;
}

void show(){
    System.out.println(id+ " "+name);
}

}
```



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.

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

</beans>
```

↑ SCROLL TO TOP

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();

    }
}
```

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">
```

↑ SCROLL TO TOP


```
<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

[download this example \(developed using Myeclipse IDE\)](#)

[download this example \(developed using Eclipse IDE\)](#)

← Prev

Next →



For Videos Join Our Youtube Channel: [Join Now](#)

↑ SCROLL TO TOP



```
1. Book.java
2.
3. Book class is a simple class consisting of the book details such title, author,
   publications and its corresponding POJO's. The getBookDetails() method will display the
   book information which is set.
4.
5. package com.javainterviewpoint;
6.
7. public class Book
8. {
9.     private String title;
10.    private String publications;
11.    private String author;
12.
13.    public Book()
14.    {
15.        super();
16.    }
17.
18.    public Book(String title, String publications, String author)
19.    {
20.        super();
21.        this.title = title;
22.        this.publications = publications;
23.        this.author = author;
24.    }
25.
26.    public String getTitle()
27.    {
28.        return title;
29.    }
30.
31.    public void setTitle(String title)
32.    {
33.        this.title = title;
34.    }
35.
36.    public String getPublications()
37.    {
38.        return publications;
39.    }
40.
41.    public void setPublications(String publications)
42.    {
43.        this.publications = publications;
44.    }
45.
46.    public String getAuthor()
47.    {
48.        return author;
49.    }
50.
51.    public void setAuthor(String author)
52.    {
53.        this.author = author;
54.    }
55.
56.    public void getBookDetails()
57.    {
58.        System.out.println("**Published Book Details**");
59.        System.out.println("Book Title      : " + title);
60.        System.out.println("Book Author    : " + author);
61.        System.out.println("Book Publications : " + publications);
62.    }
63. }
64.
65. Library.java
66.
```

67. Library class has the Book class instance as a property and its corresponding getters and setters. The book property will get its value through our configuration file.

```
68.  
69. package com.javainterviewpoint;  
70.  
71. public class Library  
72. {  
73.     private Book book;  
74.  
75.     public void setBook(Book book)  
76.     {  
77.         this.book = book;  
78.     }  
79.  
80.     public Book getBook()  
81.     {  
82.         return book;  
83.     }  
84. }
```

85. SpringConfig.xml

87.
88. In our configuration file we have defined separate id for each bean Library and Book classes

```
89.  
90. <beans xmlns="http://www.springframework.org/schema/beans"  
91. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
92. xsi:schemaLocation="http://www.springframework.org/schema/beans  
93. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
94.  
95.     <bean id="library" class="com.javainterviewpoint.Library">  
96.         <property name="book" ref="book"></property>  
97.     </bean>  
98.  
99.     <bean id="book" class="com.javainterviewpoint.Book">  
100.         <property name="title" value="Spring XML Configuration"></property>  
101.         <property name="author" value="JavaInterviewPoint"></property>  
102.         <property name="publications" value="JIP Publication"></property>  
103.     </bean>  
104. </beans>
```

105.
106. We inject primitives to the Book class properties such as title, author, publications.

```
107.  
108. <bean id="book" class="com.javainterviewpoint.Book">  
109.     <property name="title" value="Spring XML Configuration"></property>  
110.     <property name="author" value="JavaInterviewPoint"></property>  
111.     <property name="publications" value="JIP"></property>  
112. </bean>
```

113.
114. We are referencing the Book class bean id to the property book of the Library class

```
115.  
116. <property name="book" ref="book"></property>
```

117.
118. The ref passed to the property book should be the bean id of the Book Class. In short

```
119.  
120. ref =<<bean id of Book class>>  
121. Application.java
```

```
122.  
123. package com.javainterviewpoint;  
124.  
125. import org.springframework.context.ApplicationContext;  
126. import org.springframework.context.support.ClassPathXmlApplicationContext;  
127.  
128. public class Application  
129. {  
130.     public static void main(String args[])  
131.     {
```

```

132.         // Read the Spring configuration file [SpringConfig.xml]
133.         ApplicationContext appContext = new
ClassPathXmlApplicationContext("SpringConfig.xml");
134.         // Get the Library instance
135.         Library library = (Library) appContext.getBean("library");
136.         // Get the Book Details
137.         library.getBook().getBookDetails();
138.     }
139. }
140.

```

```

1. Book.java
2.
3. Book class is a simple class consisting of the book details such title, author,
publications and its corresponding POJO's. The getBookDetails() method will display the
book information which is set.
4.
5. package com.javainterviewpoint;
6.
7. public class Book
8. {
9.     private String title;
10.    private String publications;
11.    private String author;
12.
13.    public Book()
14.    {
15.        super();
16.    }
17.
18.    public Book(String title, String publications, String author)
19.    {
20.        super();
21.        this.title = title;
22.        this.publications = publications;
23.        this.author = author;
24.    }
25.
26.    public String getTitle()
27.    {
28.        return title;
29.    }
30.
31.    public void setTitle(String title)
32.    {
33.        this.title = title;
34.    }
35.
36.    public String getPublications()
37.    {
38.        return publications;
39.    }
40.
41.    public void setPublications(String publications)
42.    {
43.        this.publications = publications;
44.    }
45.
46.    public String getAuthor()
47.    {
48.        return author;
49.    }
50.
51.    public void setAuthor(String author)
52.    {
53.        this.author = author;
54.    }
55.
56.    public void getBookDetails()

```

```

57.     {
58.         System.out.println("***Published Book Details***");
59.         System.out.println("Book Title      : " + title);
60.         System.out.println("Book Author   : " + author);
61.         System.out.println("Book Publications : " + publications);
62.     }
63. }
64.
65. Library.java
66.
67. Library class has the Book class instance as a property and its corresponding getters
    and setters. The book property will get its value through our configuration file.
68.
69. package com.javainterviewpoint;
70.
71. public class Library
72. {
73.     private Book book;
74.
75.     public void setBook(Book book)
76.     {
77.         this.book = book;
78.     }
79.
80.     public Book getBook()
81.     {
82.         return book;
83.     }
84. }
85.
86. SpringConfig.xml
87.
88. In our configuration file we have defined separate id for each bean Library and Book
    classes
89.
90. <beans xmlns="http://www.springframework.org/schema/beans"
91. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
92. xsi:schemaLocation="http://www.springframework.org/schema/beans
93. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
94.
95.     <bean id="library" class="com.javainterviewpoint.Library">
96.         <property name="book" ref="book"></property>
97.     </bean>
98.
99.     <bean id="book" class="com.javainterviewpoint.Book">
100.         <property name="title" value="Spring XML Configuration"></property>
101.         <property name="author" value="JavaInterviewPoint"></property>
102.         <property name="publications" value="JIP Publication"></property>
103.     </bean>
104. </beans>
105.
106. We inject primitives to the Book class properties such as title, author,
    publications.
107.
108. <bean id="book" class="com.javainterviewpoint.Book">
109.     <property name="title" value="Spring XML Configuration"></property>
110.     <property name="author" value="JavaInterviewPoint"></property>
111.     <property name="publications" value="JIP"></property>
112. </bean>
113.
114. We are referencing the Book class bean id to the property book of the Library
    class
115.
116. <property name="book" ref="book"></property>
117.
118. The ref passed to the property book should be the bean id of the Book Class. In
    short
119.
120. ref =<<bean id of Book class>>
121. Application.java

```

```
122.
123. package com.javainterviewpoint;
124.
125. import org.springframework.context.ApplicationContext;
126. import org.springframework.context.support.ClassPathXmlApplicationContext;
127.
128. public class Application
129. {
130.     public static void main(String args[])
131.     {
132.         // Read the Spring configuration file [SpringConfig.xml]
133.         ApplicationContext appContext = new
ClassPathXmlApplicationContext("SpringConfig.xml");
134.         // Get the Library instance
135.         Library library = (Library) appContext.getBean("library");
136.         // Get the Book Details
137.         library.getBook().getBookDetails();
138.     }
139. }
140.
```