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, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

IOC- Inversion of Control - It is a programming technique which is object coupling is bound at runtime.

The binding process is achieved through dependency Injection(DI)

IOC - design principles that allow classes to be loosely coupled and therefore easier to test and maintain.

DI - Spring XML file - Beans.xml or applicationContext.xml

That will help you to execute this DI - through Bean id

IOC relies on DI

POJO - Plain Old Java Object - Attribute details and getter and setter methods

Applications of Spring

Following is the list of few of the great benefits of using Spring Framework –

- **POJO Based** - Spring enables developers to develop enterprise-class applications using POJOs. The benefit of

using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.

- **Modular** - Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- **Integration with existing frameworks** - Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- **Testablity** - Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- **Web MVC** - Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- **Central Exception Handling** - Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- **Transaction management** - Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

Benefits of Using the Spring Framework

Following is the list of few of the great benefits of using Spring Framework –

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

Dependency Injection (DI)

The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The **Inversion of Control (IoC)** is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them

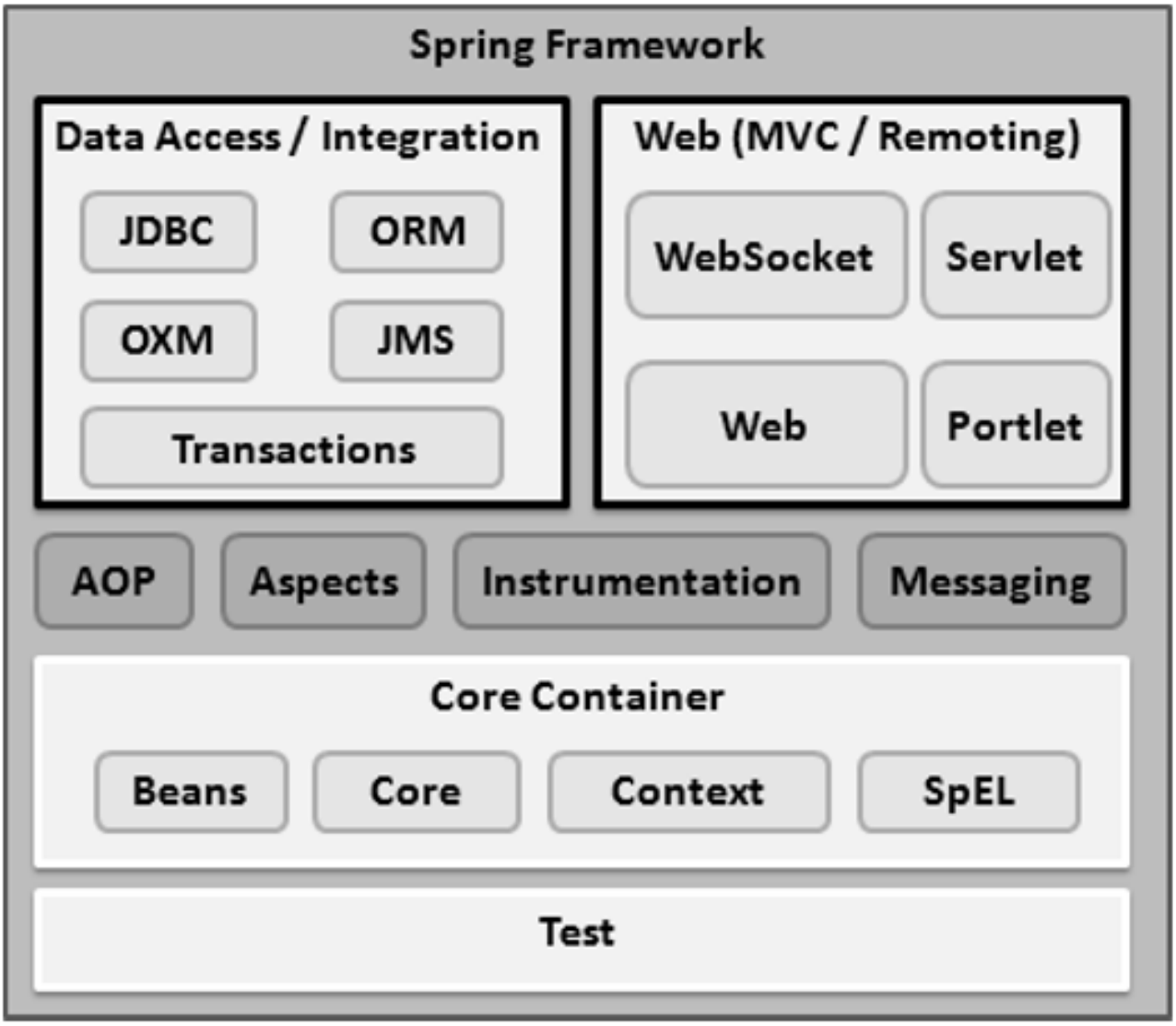
independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

Aspect Oriented Programming (AOP)

One of the key components of Spring is the **Aspect Oriented Programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.

The AOP module of Spring Framework 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 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.

What is the Spring Container?

The Spring container is responsible for instantiating, configuring, and assembling the Spring beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich inter-dependencies between those objects.

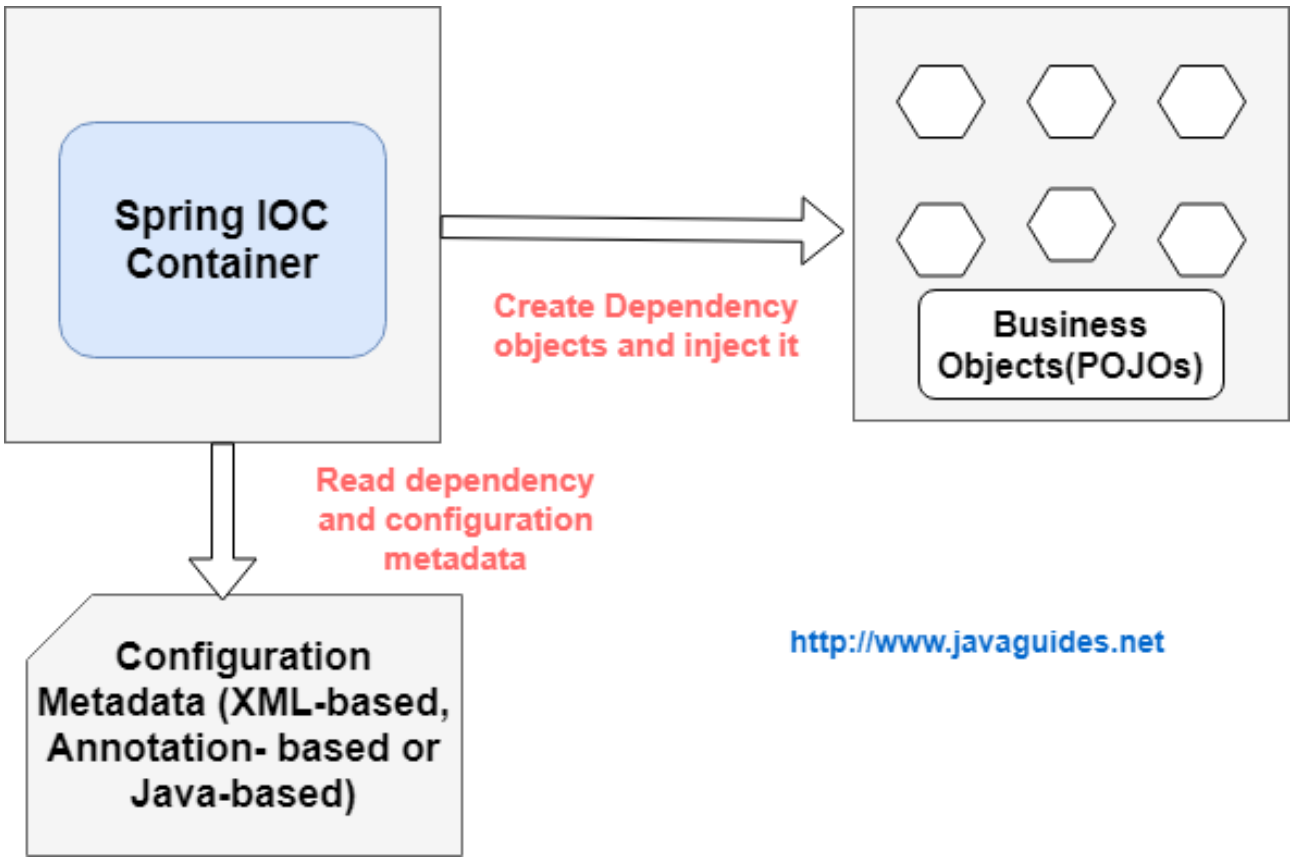
The responsibilities of IOC container are:

- Instantiating the bean
- Wiring the beans together
- Configuring the beans
- Managing the bean's entire life-cycle

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. Spring framework provides two distinct types of containers.

1. `BeanFactory` container
2. `ApplicationContext` container

`BeanFactory` is the root interface of Spring IOC container. `ApplicationContext` is the child interface of `BeanFactory` interface that provides Spring AOP features, i18n etc.
One main difference between `BeanFactory` and `ApplicationContext` is that `BeanFactory` only instantiates bean when we call `getBean()` method while `ApplicationContext` instantiates singleton bean when the container is started, It doesn't wait for `getBean()` method to be called.



What is Configuration Metadata?

From the above diagram, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.
Three ways we can supply Configuration Metadata to Spring IoC container

1. XML-based configuration
2. Annotation-based configuration
3. Java-based configuration

How to Create a Spring Container?

Spring provides many ApplicationContext interface implementations that we use are;

1. `AnnotationConfigApplicationContext`: If we are using Spring in standalone Java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects. - @Bean

2. `ClassPathXmlApplicationContext`: If we have spring bean configuration XML file in a standalone application, then we can use this class to load the file and get the container object.
3. `FileSystemXmlApplicationContext`: This is similar to `ClassPathXmlApplicationContext` except that the XML configuration file can be loaded from anywhere in the file system.

`AnnotationConfigWebApplicationContext` and `XmlWebApplicationContext` for web applications.
Let's write a code to create Spring container:

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

Note that we are supplying configuration metadata via applicationContext.xml file(XML-based configuration).

```
AnnotationConfigApplicationContext  context = new
AnnotationConfigApplicationContext(AppConfig.class);
```

Note that we are supplying configuration metadata via AppConfig.class file.

The most used API that implements the BeanFactory is the XmlBeanFactory.

```
 XmlBeanFactory factory = new XmlBeanFactory (new
ClassPathResource("applicationContext.xml"));
```

Next, how to Retrieve bean from spring container?

How to Retrieve Bean from Spring Container?

Both `BeanFactory` and `ApplicationContext` interface provides `getBean()` method to retrieve bean from spring container.

ApplicationContext getBean() Example:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
HelloWorld obj = (HelloWorld)
context.getBean("helloWorld");
```

BeanFactory getBean() Example:

```
XmlBeanFactory factory = new XmlBeanFactory (new
ClassPathResource("beans.xml"));
HelloWorld obj = (HelloWorld)
factory.getBean("helloWorld");
```

Spring IOC Container XML Config Example

Spring Application Development Steps

Follow these three steps to develop a spring application:

1. Create a simple Maven Project
2. Add Maven Dependencies
3. Configure `HellowWorld` Spring Beans
4. Create a Spring Container
5. Retrieve Beans from Spring Container

Tools and technologies used

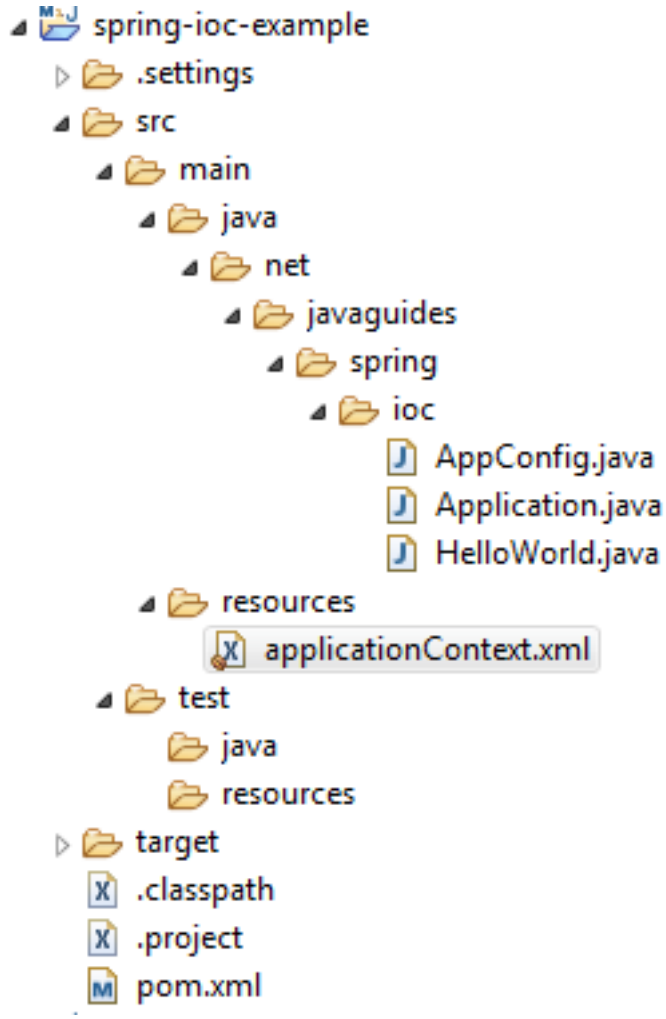
- Spring Framework - 5.1.0.RELEASE
- JDK - 8 or later
- Maven - 3.2+
- IDE - Eclipse Mars/STS

1. Create a simple Maven Project

Create a simple maven project using your favorite IDE and refer below diagram for packaging structure. If you are new to maven then read this article **How to Create a Simple Maven Project**.

Project Structure

Below diagram shows a project structure for your reference -



2. Add Maven Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.spring</groupId>
    <artifactId>spring-ioc-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</
project.build.sourceEncoding>
    </properties>

    <dependencies>
```

```
        <!-- https://mvnrepository.com/artifact/
org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.0.RELEASE</version>
        </dependency>
    </dependencies>
    <build>
        <sourceDirectory>src/main/java</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</
artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

3. Configure HelloWorld Spring Beans

What Is a Spring Bean?

This is a very simple question that is often overcomplicated. Usually, Spring beans are Java objects that are managed by the Spring container.
Here is a HelloWorld Spring bean:

```
package com.spring.spring_ioc_example;

public class HelloWorld {
    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public void getMessage() {
        System.out.println("My Message : " + message);
    }
```

```
}
```

Configuration Metadata - Configure HelloWorld Spring Beans

```
<?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="helloWorld"
class="com.spring.spring_ioc_example.HelloWorld">
  <property name="message" value="Hello World!" />
 </bean>
</beans>
```

4. Create a Spring Container

If we have spring bean configuration XML file in a standalone application, then we can use ClassPathXmlApplicationContext class to load the file and get the container object.

```
package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    }
}
```

5. Retrieve Beans from Spring Container

`ApplicationContext` interface provides `getBean()` method to retrieve bean from spring container.

```
package net.javaguides.spring.ioc;
```

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        HelloWorld obj = (HelloWorld)
context.getBean("helloWorld");
        obj.getMessage();
    }
}
```

Output

```
My Message : Hello World!
```

Spring IOC Container XML Config Example

Spring Application Development Steps

Follow these three steps to develop a spring application:

1. Create a simple Maven Project
2. Add Maven Dependencies
3. Configure `HellowWorld` Spring Beans
4. Create a Spring Container
5. Retrieve Beans from Spring Container

Tools and technologies used

- Spring Framework - 5.1.0.RELEASE
- JDK - 8 or later
- Maven - 3.2+
- IDE - Eclipse Mars/STS

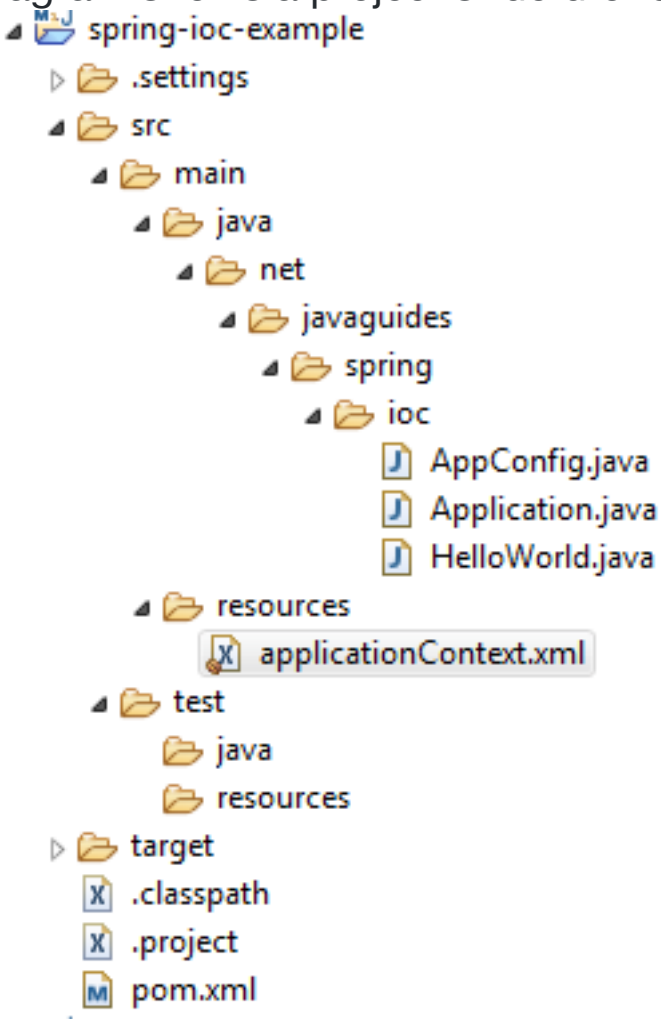
1. Create a simple Maven Project

Create a simple maven project using your favorite IDE and refer below diagram for packaging structure. If you are

new to maven then read this article **How to Create a Simple Maven Project**.

Project Structure

Below diagram shows a project structure for your reference -



2. Add Maven Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>net.javaguides.spring</groupId>
    <artifactId>spring-ioc-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <url>http://maven.apache.org</url>

    <properties>
```

```
        <project.build.sourceEncoding>UTF-8</
project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/
org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.0.RELEASE</version>
        </dependency>
    </dependencies>
    <build>
        <sourceDirectory>src/main/java</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</
artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

3. Configure HelloWorld Spring Beans

What Is a Spring Bean?

This is a very simple question that is often overcomplicated. Usually, Spring beans are Java objects that are managed by the Spring container.
Here is a HelloWorld Spring bean:

```
package net.javaguides.spring.ioc;

public class HelloWorld {
    private String message;

    public void setMessage(String message) {
        this.message = message;
```

```
    }

    public void getMessage() {
        System.out.println("My Message : " + message);
    }
}
```

Configuration Metadata - Configure HelloWorld Spring Beans

```
<?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="helloWorld"
class="net.javaguides.spring.ioc.HelloWorld">
  <property name="message" value="Hello World!" />
 </bean>
</beans>
```

4. Create a Spring Container

If we have spring bean configuration XML file in a standalone application, then we can use ClassPathXmlApplicationContext class to load the file and get the container object.

```
package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    }
}
```

5. Retrieve Beans from Spring Container

`ApplicationContext` interface provides `getBean()` method to retrieve bean from spring container.

```
package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        HelloWorld obj = (HelloWorld)
context.getBean("helloWorld");
        obj.getMessage();
    }
}
```

Output

```
My Message : Hello World!
```

What is ApplicationContext Interface?

The `ApplicationContext` is the central interface within a Spring application for providing configuration information to the application.

The interfaces `BeanFactory` and `ApplicationContext` represent the Spring IoC container. Here, `BeanFactory` is the root interface for accessing the Spring container. It provides basic functionalities for managing beans.

On the other hand, the `ApplicationContext` is a sub-interface of the `BeanFactory`. Therefore, it offers all the functionalities

of `BeanFactory`. Furthermore, it provides more enterprise-specific functionalities.

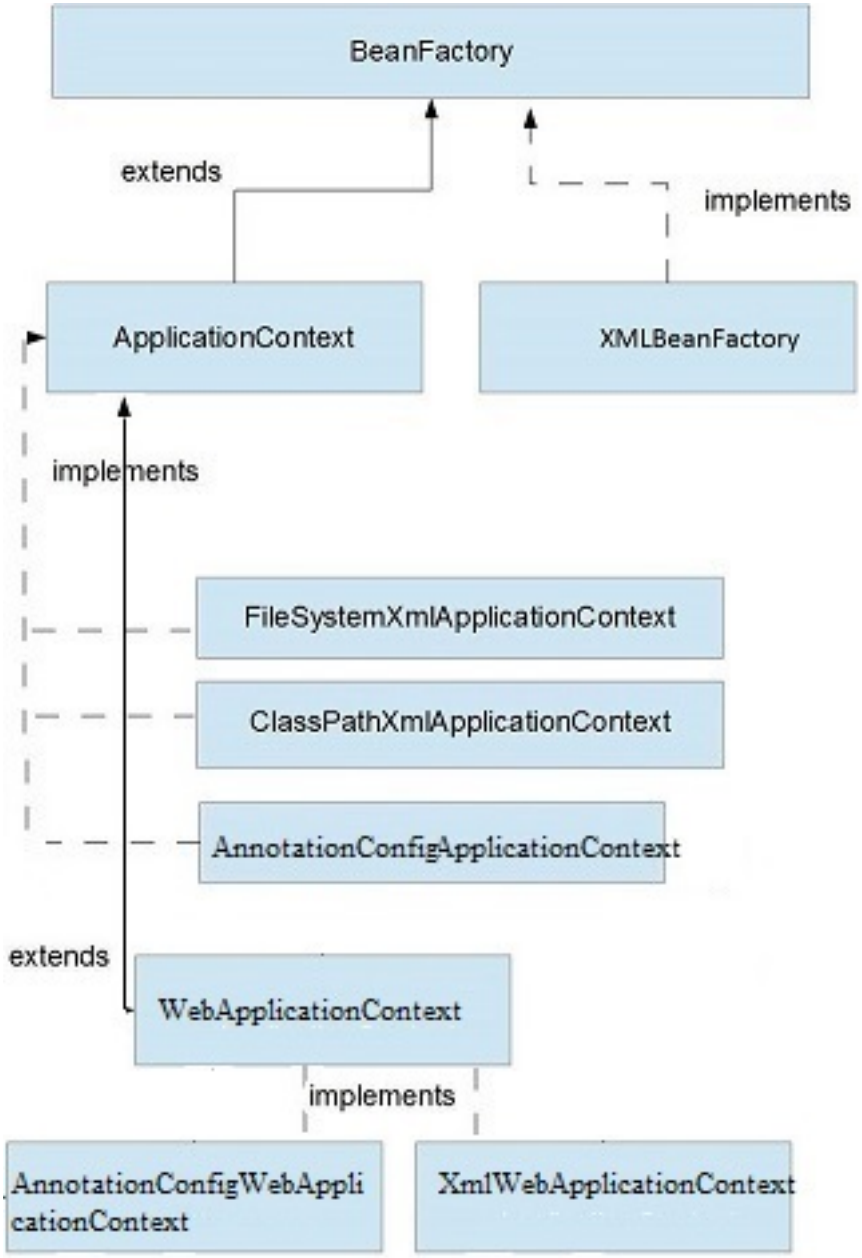
The important features of `ApplicationContext` are

- resolving messages
- supporting internationalization,
- publishing events
- application-layer-specific contexts

This is why we use `ApplicationContext` as the default Spring container.

ApplicationContext Interface Implementation Classes

The below diagram shows the implementations of `BeanFactory` and `ApplicationContext` interfaces:



Let's take a look into all the `ApplicationContext` Interface implementation classes.

1. FileSystemXMLApplicationContext

We use the **FileSystemXMLApplicationContext** class to load an XML-based Spring configuration file from the file system or from URLs.

For example, let's see how we can create this Spring container and load the beans for our XML-based configuration:

```
String path = "C:/Spring-demo/src/main/resources/spring-servlet.xml";
```

```
ApplicationContext appContext = new FileSystemXmlApplicationContext(path);
AccountService accountService = appContext.getBean("studentService",
StudentService.class);
```

2. ClassPathXmlApplicationContext

In case we want to load an XML configuration file from the classpath, we can use
the **ClassPathXmlApplicationContext** class.

Similar to `FileSystemXMLApplicationContext`, it's useful for test harnesses, as well as application contexts embedded within JARs.

For example:

```
ApplicationContext appContext = new ClassPathXmlApplicationContext("spring-
servlet.xml");
AccountService accountService = appContext.getBean("studentService",
StudentService.class);
```

3. XmlWebApplicationContext

If we use the XML-based configuration in a web application, we can use the **XmlWebApplicationContext** class.

Configuration classes declared and typically loaded from XML file in `/WEB-INF/`

For example:

```
public class MyXmlWebApplicationInitializer implements
WebApplicationInitializer {

  public void onStartup(ServletContext container) throws ServletException {
    XmlWebApplicationContext context = new XmlWebApplicationContext();
    context.setConfigLocation("/WEB-INF/spring/applicationContext.xml");
    context.setServletContext(container);

    // Servlet configuration
  }
}
```

4. AnnotationConfigApplicationContext

The **AnnotationConfigApplicationContext** class was introduced in Spring 3.0. It can take classes annotated with @Configuration, @Component, and JSR-330 metadata as input.

So let's see a simple example of using the AnnotationConfigApplicationContext container with our Java-based configuration:

```
ApplicationContext appContext = new
AnnotationConfigApplicationContext(StudentConfig.class);
AccountService accountService = appContext.getBean(StudentService.class);
```

5. AnnotationConfigWebApplicationContext

The **AnnotationConfigWebApplicationContext** is a web-based variant of `AnnotationConfigApplicationContext`.

We may use this class when we configure Spring's ContextLoaderListener servlet listener or a Spring MVC DispatcherServlet is in a `web.xml` file.

Moreover, from Spring 3.0 onward, we can also configure this application context container programmatically. All we need to do is implement the **WebApplicationInitializer** interface:

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

  public void onStartup(ServletContext container) throws ServletException {
    AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
    context.register(StudentConfig.class);
    context.setServletContext(container);

    // servlet configuration
  }
}
```

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled

Dependency Lookup

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

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

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

Dependency Injection

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

```
class Employee{
Address address;

Employee(Address address){
this.address=address;
}
public void setAddress(Address address){
```

```
this.address=address;
}

}
```

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

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

```
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 Employee(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>
```

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();

    }
}
```

Output:10 null

```
<bean id="e" class="com.spring.DI.Employee">
<constructor-arg value="Bhuvana"></constructor-arg>
</bean>
```

Output: 0 Bhuvana

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="Bhuvana"></constructor-arg>
</bean>
```

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.

```
<groupId>com.spring.Constructor.DO</
groupId>
  <artifactId>Spring-Constructor-DO</
artifactId>

WRITE THE DEPENDENCY IN pom.xml
```

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

```
package com.spring.constructor.DO

public class Address {
private String city;
private String state;
private String country;

public Address(String city, String state, String country) {
    super();
    this.city = city;
    this.state = state;
    this.country = country;
```

```
}

public String toString(){
    return city+" "+state+" "+country;
}
}
```

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.

```
package com.spring.constructor.DO
public class Employee {
private int id;
private String name;
private Address address;//Aggregation

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

public Employee(int id, String name, Address address) {
    super();
    this.id = id;
    this.name = name;
    this.address = address;
}

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

}
```

Beans.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.spring.constructor.DO.Address”>
<constructor-arg value="Chennai"></constructor-arg>
<constructor-arg value="TN"></constructor-arg>
<constructor-arg value="India"></constructor-arg>
</bean>


<bean id="e" class=“com.spring.constructor.DO.Employee">
<constructor-arg value="12" type="int"></constructor-arg>
<constructor-arg value="Bhuvana"></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.spring.constructor.DO

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("Beans.xml");
        BeanFactory factory=new XmlBeanFactory(r);

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

    }
}
```

Output:

12 Bhuvana

Chennai TN India

Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the **constructor-arg** element.

It can be:

1. **list**
2. **set**
3. **map**

Each collection can have string based and non-string based values.

In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages:

1. **Question.java**
2. **applicationContext.xml**
3. **Test.java**

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

```
<groupId>com.spring.constructor.collecti
on</groupId>
  <artifactId>Spring-Constructor-
Collection</artifactId>
```

Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
package com.spring.constructor.collection;

import java.util.Iterator;
import java.util.List;

public class Question {
private int id;
private String name;
private List<String> answers;

public Question() {}
public Question(int id, String name, List<String> answers) {
   super();
   this.id = id;
   this.name = name;
   this.answers = answers;
}
```

```
public void displayInfo(){
    System.out.println(id+" "+name);
    System.out.println("answers are:");
    Iterator<String> itr=answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}

}
```

applicationContext.xml
The list element of constructor-arg is used here to define the list.

```
<?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="q" class="com.spring.constructor.collection.Question">

<constructor-arg value="111"></constructor-arg>
<constructor-arg value="What is java?"></constructor-arg>
<constructor-arg>
<list>
<value>Java is a programming language</value>
<value>Java is a Platform</value>
<value>Java is an Island of Indonasia</value>
</list>
</constructor-arg>
</bean>

</beans>
```

Test.java
This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
package com.spring.constructor.collection;

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;

public class Test {
public static void main(String[] args) {
     Resource r=new ClassPathResource("applicationContext.xml");

    BeanFactory factory=new XmlBeanFactory(r);

    Question q=(Question)factory.getBean("q");
    q.displayInfo();

}
}
```

Output:

```
111 What is java?
answers are:
Java is a programming language
Java is a Platform
Java is an Island of Indonesia
```

Constructor Injection with Map Example

In this example, we are using **map** as the answer that have answer with posted username. Here, we are using key and value pair both as a string.

Like previous examples, it is the example of forum where **one question can have multiple answers**.

Question.java
This class contains three properties, two constructors and displayInfo() method to display the information.

```
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;

public class Question {
private int id;
private String name;
private Map<String,String> answers;

public Question() {}
public Question(int id, String name, Map<String, String> answers)
{
    super();
    this.id = id;
    this.name = name;
    this.answers = answers;
}

public void displayInfo(){
    System.out.println("question id:"+id);
    System.out.println("question name:"+name);
    System.out.println("Answers....");
    Set<Entry<String, String>> set=answers.entrySet();
    Iterator<Entry<String, String>> itr=set.iterator();
    while(itr.hasNext()){
```

```
        Entry<String,String> entry=itr.next();
                                System.out.println("Answer:"+entry.getKey()
+" Posted By:"+entry.getValue());
    }
}
}
```

Beans.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"
    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="q" class="com.javatpoint.Question">
<constructor-arg value="11"></constructor-arg>
<constructor-arg value="What is Java?"></constructor-arg>
<constructor-arg>
<map>
<entry key="Java is a Programming Language"  value="Ajay Kuma
r"></entry>
<entry key="Java is a Platform" value="John Smith"></entry>
<entry key="Java is an Island" value="Raj Kumar"></entry>
</map>
</constructor-arg>
</bean>

</beans>
```

Test.java

```
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;
```

```
public class Test {
public static void main(String[] args) {
     Resource r=new ClassPathResource("applicationContext.xml");

    BeanFactory factory=new XmlBeanFactory(r);

    Question q=(Question)factory.getBean("q");
    q.displayInfo();

}
}
```

Dependency Injection by setter method

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

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

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

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}

public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
void display(){
    System.out.println(id+" "+name+" "+city);
}

}
```

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.

```
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

Setter Injection with Dependent Object Example

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.

```
public class Address {
private String addressLine1,city,state,country;

//getters and setters

public String toString(){
    return addressLine1+" "+city+" "+state+" "+country;
}
```

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

```
public class Employee {
private int id;
private String name;
private Address address;

//setters and getters

void displayInfo(){
```

```
    System.out.println(id+" "+name);
    System.out.println(address);
}
}
```

applicationContext.xml

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

```
<?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="address1" class="com.javatpoint.Address">
<property name="addressLine1" value="51,Lohianagar"></
property>
<property name="city" value="Ghaziabad"></property>
<property name="state" value="UP"></property>
<property name="country" value="India"></property>
</bean>

<bean id="obj" class="com.javatpoint.Employee">
<property name="id" value="1"></property>
<property name="name" value="Sachin Yadav"></property>
<property name="address" ref="address1"></property>
</bean>

</beans>
```

Test.java

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

```
package com.javatpoint;
```

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import  org.springframework.context.support.ClassPathXmlApplica
tionContext;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

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");
    e.displayInfo();

}
}
```

Setter Injection with Collection Example

We can inject collection values by setter method in spring framework. There can be used three elements inside the **property** element.

It can be:

1. **list**
2. **set**
3. **map**

Each collection can have string based and non-string based values.

In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages: