Spring Basics and Inversion Of Control – I

Core Spring

Harish B Rao
Talent Transformation | Wipro Technologies

Core Spring

The Core Spring can be thought of as a ***Framework*** and a ***Container*** for managing ***Business Objects*** and their relationship. When we use this framework, we don't need to depend on Spring specific Classes and Interfaces. This is unlike other frameworks, where they will force the Client Applications to depend on their propriety Implementations.

For example, consider the various J2EE Components like ***Servlets*** or ***EJB.*** If a developer wants to write a Servlet, the class has to depend on HttpServlet, same is the case of creating Enterprise Java Beans(Version 2.x).

Core Spring

The architects of Spring have spent enough time in designing the framework to keep the coupling between the Clients and the ***Spring Framework*** to a bare minimum. In most cases, the coupling is often nil.

In other words, Business Components you write in Spring are ***POJO***s (Plain Old Java Object) or ***POJI*** s(Plain Old Java Interface) only. POJO/POJI refers to Classes or Interfaces that do not specially extend or implement third-party Implementations. The main advantage of having most of the Classes or Interfaces as POJO/POJI in an Application is that, they will facilitate easy ***Unit Testing*** in the Application.

Spring Core API

The ***Core API*** in Spring is very limited and it generally involves in :

- ***Configuring***
- ***Creating***
- ***Making Associations***

between various ***Business Components***.

Spring refers to these Business Components as ***Beans***.

Spring Container

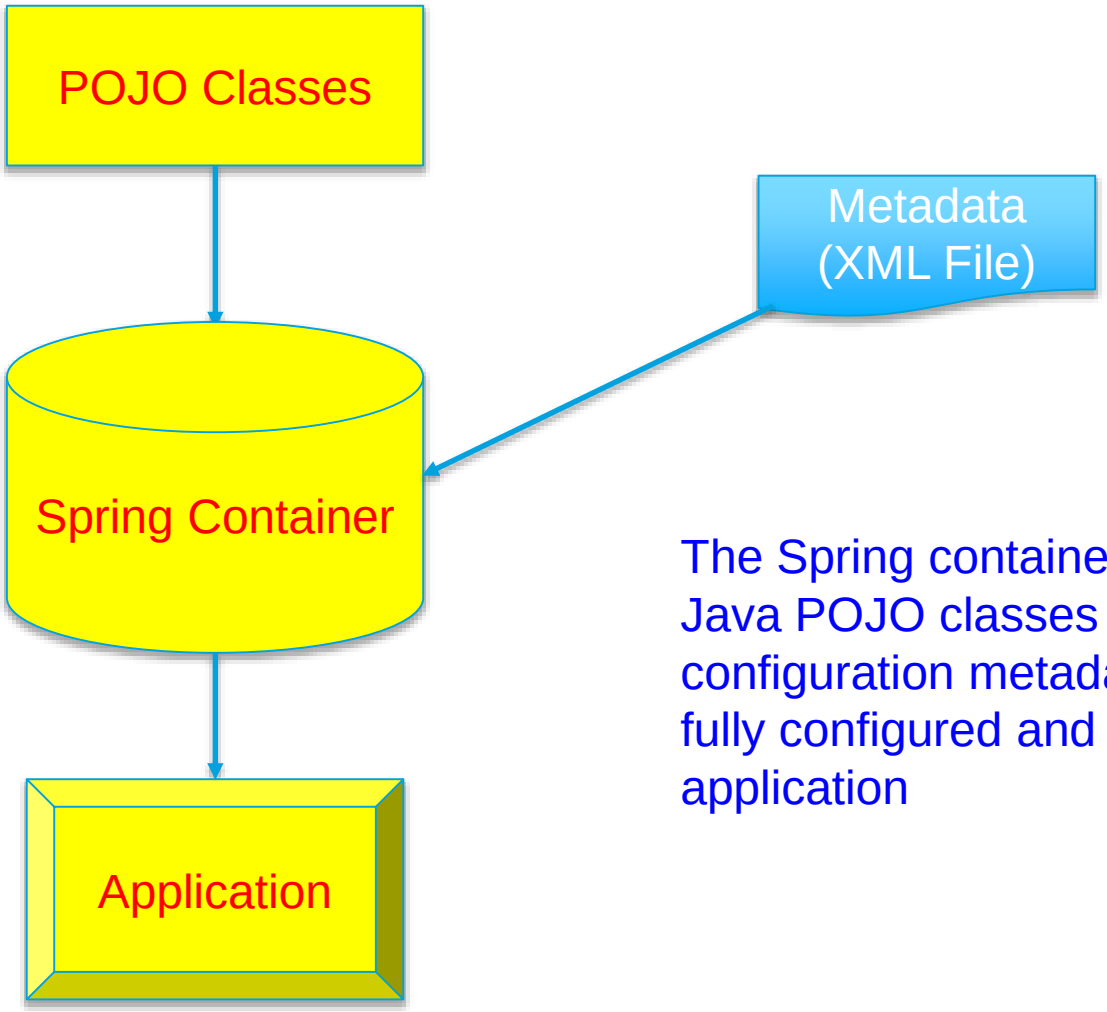
The spring container is at the core of Spring Framework.

The container is responsible for creating the objects, wiring them together, configuring them and managing their life cycle from creation to destruction.

The spring container uses Dependency Injection to manage the bean components and these components are assembled to create an application.

*** If you are wondering what is wiring, configuring objects, dependency injection etc… wait! You will know, by the end of this module..☺*

Spring Container (Contd.).



The Spring container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable application

Spring Container (Contd.).

The container reads the configuration metadata to find out what objects to instantiate, configure and assemble.

The configuration metadata can be stored either in an XML file or Java code.

We can also provide metadata through annotations.

The following are the Interfaces that are available in Spring for achieving this goal :

- **Resource**
- **BeanFactory** & **ApplicationContext**

Resource

Resource interface is used for abstracting access to low-level resources.

In other words, we can create an object of the type Resource, which can represent a file or a class path resource.

This interface is available in the package :
org.springframework.core.io

Some of the classes which provide concrete implementation of 'Resource' interface are:

- FileSystemResource
- ClassPathResource
- InputStreamResource

Resource (Contd.).

A Resource in Spring represents any kind of Information that comes from a ***File*** or a ***Stream***.

For example, resource could represent an Xml File containing the various Configuration Information needed for a Spring Application or it could represent a ***Java Class File*** representing a Bean object.

Whatever be the case, it can be represented as a Resource object with the transparent nature of its implementation.

Suppose, we wish to load a Resource that represents the Java Class called Mumbai, then we can create a Resource object in the following manner :

```
Resource classRes = new ClassPathResource("PathToClassFile",
Mumbai.class);
```

Note that ClassPathResource is the concrete implementation class for loading a Java Class file.

Resource (Contd.).

The following code loads an Xml File from the local File System

```
String xmlFile = "spring.xml";
Resource xmlResource = new FileSystemResource(xmlFile);
```

Note that FileSystemResource can be used to load any kind of files to make themselves available to the Spring Application. It is not restricted to only Xml Files.

Other commonly used Resource is the InputStreamResource, that loads content from an Input Stream.

Many concrete implementations of various Resources are available in Spring Framework.

BeanFactory & ApplicationContext

Spring Framework provides two distinct types of containers. They are

- **BeanFactory**
- **ApplicationContext**

Two of the most fundamental and important packages in Spring are:

- **org.springframework.beans**
- **org.springframework.context**

Classes and Interfaces in these package hierarchy provide the basis for Spring's *Inversion of Control* – **IOC** features

(Inversion of Control is explained in detail, later in the course)

BeanFactory & ApplicationContext (Contd.).

The BeanFactory provides the configuration framework and basic functionality.

The ApplicationContext adds enhanced capabilities to it, some of them more J2EE and enterprise-centric.

The functionality and the usage of these objects are described in detail, in the next few slides.

Bean Factory

- The **BeanFactory** provides an advanced configuration mechanism capable of managing beans (objects) of any nature

- The **BeanFactory** is the actual ***container*** which instantiates, configures, and manages a number of beans
  - These beans typically collaborate with one another and thus have dependencies between themselves
  - These dependencies are reflected in the configuration data used by the BeanFactory

- A BeanFactory is represented by the interface **org.springframework.beans.factory.BeanFactory**, for which there are multiple implementations.
  - The most commonly used simple BeanFactory implementation is org.springframework.beans.factory.xml.**XmlBeanFactory**

Bean Factory (Contd.).

As you are already aware, in Spring terminology a ***Bean*** refers to a ***Business Component*** in consideration.

As such, BeanFactory is the factory class for creating Bean objects.

The point of interest here is, how to configure the BeanFactory for creating Business Components. In other words, Where exactly should the BeanFactory object look for Bean definitions? Or Where are the Bean definitions required for creating Bean instances?

One important thing to note here is that, all Beans that reside in the ***Context of Spring Container*** are highly configurable through external files. It means that your Bean definitions can reside in an Xml File, a Java Property File or even in a database.

Continued…

Bean Factory (Contd.).

Although, Bean definitions are not tightly coupled to any format, most developers prefer having their Bean definitions in Xml Format.

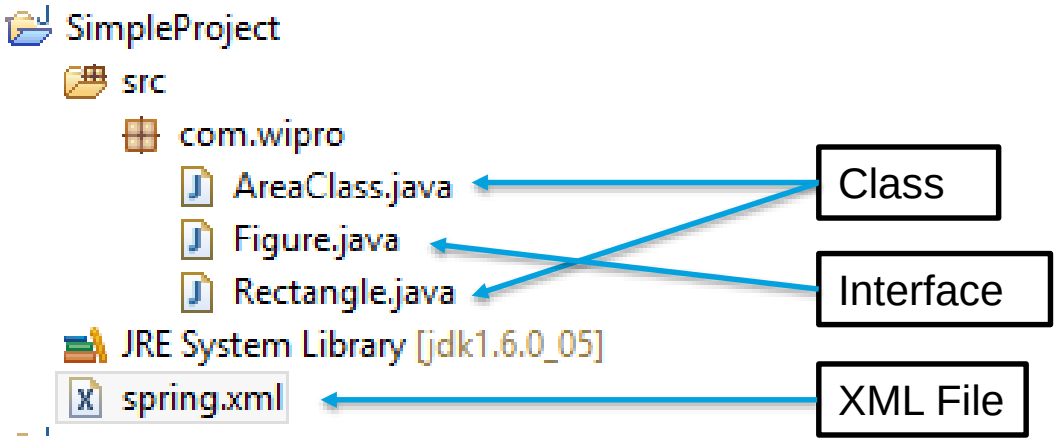
Following is the code that will load all Bean definitions from an Xml File.

```
Resource xmlResource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(xmlResource);
```

The XmlBeanFactory class is one of the concrete implementations of BeanFactory interface.

A simple Example that uses BeanFactory

Using Eclipse, let us create a Java Project called SimpleProject.

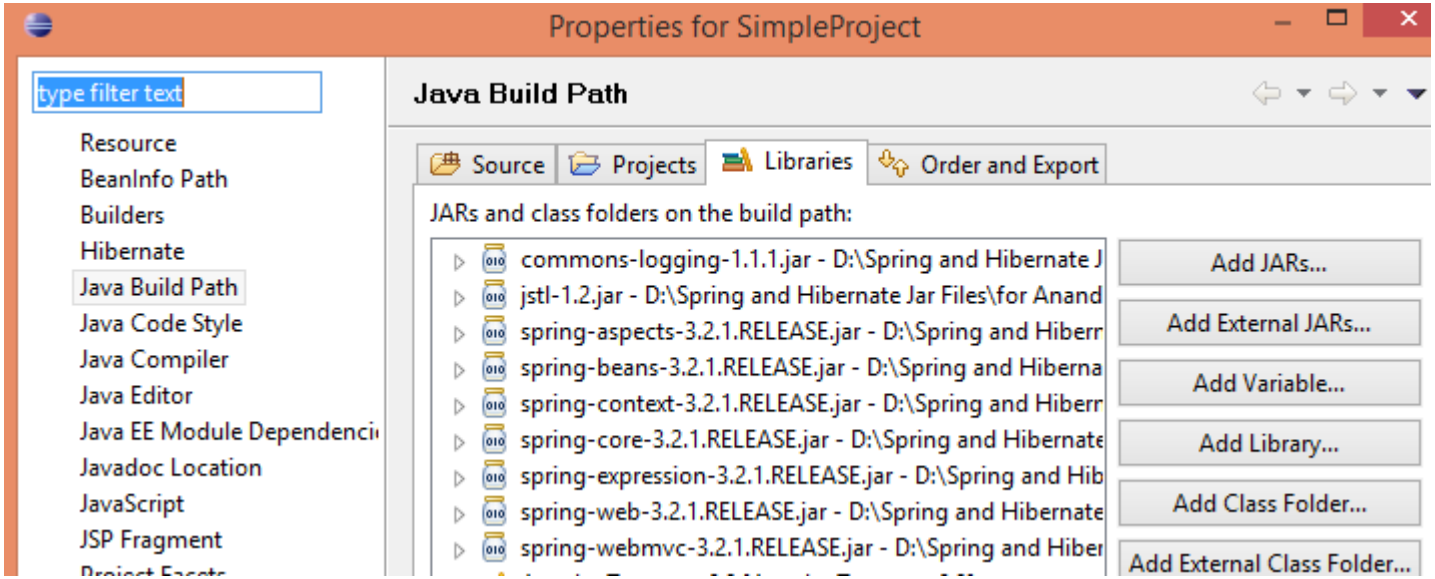


Create a package named com.wipro and create two classes AreaClass and Rectangle within this package. Also create an interface named Figure within this package.

Create an xml file with any name(here this xml file is named as spring.xml) directly under the project, as shown in the figure above.

A simple example (Contd.).

Jar files :



Add the jar files which are listed above, to the build path. You can obtain the above jar files from your faculty/facilitator.

A simple example (Contd.).

Figure.java

```
package com.wipro;

public interface Figure {
	void area();
}
```

Rectangle.java

```
package com.wipro;

public class Rectangle implements Figure {
	double d1=10;
	double d2=20;
	public void area() {
		System.out.println("The area of Rectangle is " + (d1*d2));
	}
}
```

A simple example (Contd.).

AreaClass.java

```
package com.wipro;

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

public class AreaClass {
    public static void main(String[] args) {
        Resource xmlResource = new FileSystemResource("spring.xml");
        BeanFactory bf = new XmlBeanFactory(xmlResource);
        Figure fig = (Figure)bf.getBean("fig");
        fig.area();
    }
}
```

Here, the AreaClass instantiates an object of the type, BeanFactory, using one of its implementations, XMLBeanFactory. The configuration information is stored in the xml file, spring.xml, which is used by the BeanFactory object to get the bean details.

A simple example (Contd.).

spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="fig" class="com.wipro.Rectangle"  />
</beans>
```

*** Point to remember : Place this xml file directly under the project folder(right click the project folder, SimpleProject and create this file)*

Execute this code and observe the result. It should print the following output on the console screen :

The area of Rectangle is 200.0

Understanding the demo example

```
public static void main(String[] args) {
        Resource xmlResource = new FileSystemResource("spring.xml");     // Line 1
        BeanFactory bf = new XmlBeanFactory(xmlResource);                // Line 2
        Figure fig = (Figure)bf.getBean("fig");                          // Line 3
        fig.area();                                                      // Line 4
    }
```

Let us now try to understand the code within main method of AreaClass :

Line 1: We instantiate an object of FileSystemResource and while doing so, we pass a String argument to the constructor. This string represents an xml file on the disk, “spring.xml”. If no path is specified, the spring container tries to locate this file within the root of the application. Remember, this object is of type Resource.

Line 2 : Next, we instantiate an object of XmlBeanFactory, which is a class that implements BeanFactory interface. Notice that this object is of type BeanFactory. We pass the object of type Resource, created in line 1, as an argument to the constructor of XmlBeanFactory. Thus, the BeanFactory object knows from where to read the configuration details.

Line 3 : Next, we create an object of the type Figure by invoking getBean() method on the BeanFactory object(bf.getBean()). We pass a String”fig”, as an argument to the getBean method. Now, what is this “fig”? What does this string represent? The container will read the configuration file(spring.xml) and will know that “fig” is the identifier for a bean that represents com.wipro.Rectangle object. So, the container will instantiate this class and inject this object into the client code.

Line 4 : Once the Figure object is available, the area method is invoked on the Figure object.

(If you are wondering, what does “injecting an object” mean, refer to Dependency Injection module available later in this module)

Sample Application

The participants are expected to try out this example.

In this simple application, which is based on “Spring”, we create the following:

- **Business Object** (Namer.java)

  Business Component ‘Namer’, which is used to store the given name

- **XML Configuration file** (namer.xml)

  To define and configure the Bean class along with its properties

- **Client program** (SimpleSpringApp.java)

  Which makes reference to the Xml File using the Resource object and then the contents of the Xml File are read using the object of the type BeanFactory

  An instance of the object of type Namer is then retrieved, by calling the BeanFactory.getBean(id) method

Sample Application (Contd.).

Namer.java

```
public class Namer {
    private String name;
    public Namer() {
    }
    public String getName() {
            return name;
    }
    public void setName(String name) {
            this.name = name;
    }
}
```

Sample Application (Contd.).

namer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
      <bean id="namerId" class="Namer" >
            <property name="name" >
                  <value>Abhishek</value>
            </property>
      </bean>
</beans>
```

Sample Application (Contd.).

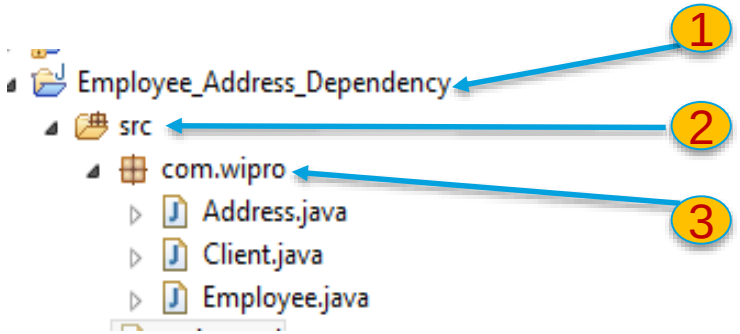
SimpleSpringApp.java (Client code)

```
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
import org.springframework.core.io.*;

public class SimpleSpringApp {
    public static void main(String args[]){
      Resource namerXmlFile = new FileSystemResource("src/namer.xml");
      BeanFactory factory = new XmlBeanFactory(namerXmlFile);
      Namer namer = (Namer)factory.getBean("namerId");
      System.out.println(namer.getName());
    }
}
```

Quiz

Given the folder structure



And the following client code :

```
public static void main(String[] args) {
        Resource xmlResource = new FileSystemResource("spring.xml");
        BeanFactory bf = new XmlBeanFactory(xmlResource);
        ……..
}
```

Within which folder you should create the file spring.xml ?

1. Folder Employee_Address_Dependency(project folder) represented by arrow 1
2. Folder src represented by arrow 2.
3. Package com.wipro represented by arrow 3.
4. None of the above

ApplicationContext

The Application Context is spring's more advanced container.

It is similar to BeanFactory. It can load bean definitions, wire beans together and dispense beans upon request.

Additionally it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file etc.

This container is defined by the org.springframework.context.ApplicationContext interface.

The ApplicationContext includes all the functionalities of BeanFactory and it is generally recommended over the BeanFactory.

BeanFactory can still be used for light weight applications like mobile devices or applet based applications.

ApplicationContext

The most commonly used ApplicationContext implementations are:

FileSystemXmlApplicationContext : This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.

ClassPathXmlApplicationContext : This container loads the definitions of the beans from an XML file. Here you don't have to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look for bean configuration XML file in CLASSPATH.

WebXmlApplicationContext : This container loads the XML file with definitions of all beans from within a web application.

BeanFactory Vs ApplicationContext

Users are sometimes unsure, what is best suited in a particular situation (whether to use a BeanFactory or an ApplicationContext).

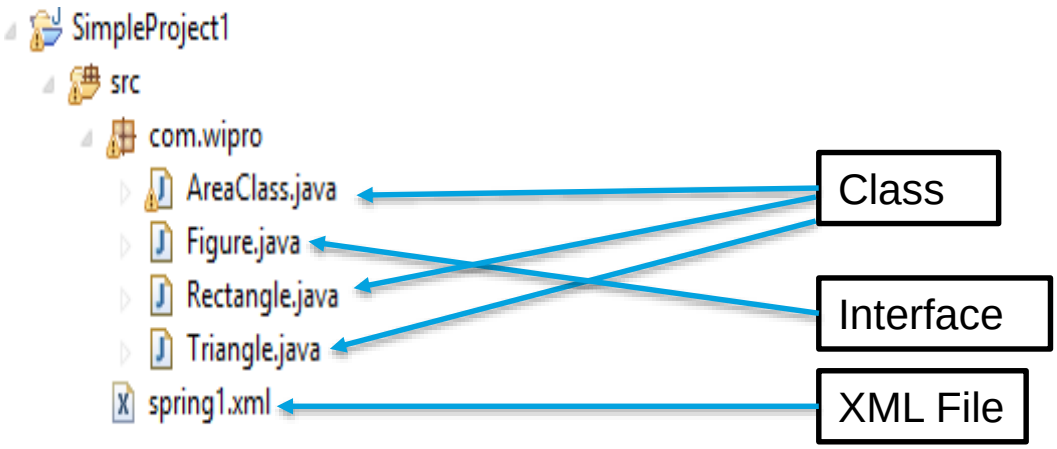
The best option is to use an ApplicationContext, when building applications in a J2EE environment. It offers all the features of the BeanFactory and adds on to it in terms of features, while allowing more declarative approach to be used in some of the functionalities.

The main usage scenario where you might prefer to use the BeanFactory is, when the memory usage is the greatest and you don't need all the features of the ApplicationContext.

To summarize, the ApplicationContext is generally recommended over the BeanFactory as it includes all functionalities of the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications.

A simple Example that uses ApplicationContext

Using Eclipse, let us create a Java Project called SimpleProject1.



Create a package named com.wipro and create three classes AreaClass, Rectangle and Triangle within this package. Also create an interface named Figure within this package.

Create an xml file with any name(here this xml file is named as spring1.xml) under the src folder, as shown in the figure above.

Example that uses ApplicationContext (Contd.).

Figure.java

```
package com.wipro;
public interface Figure {
	void area();
}
```

Rectangle.java

```
package com.wipro;
public class Rectangle implements Figure {
	double d1=10, d2=20;
	public void area() {
		System.out.println("The area of Rectangle is " + (d1*d2));
	}
}
```

Triangle.java

```
package com.wipro;
public class Triangle implements Figure {
	double d1=10, d2=20;
	public void area() {
		System.out.println("The area of triangle is " + (0.5*d1*d2));
	}
}
```

Example that uses ApplicationContext (Contd.).

AreaClass.java

```
package com.wipro;

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

public class AreaClass {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring1.xml");
         Figure fig = (Figure)context.getBean("fig");
         fig.area();
    }
}
```

Here, the AreaClass instantiates an object of the type ApplicationContext, using one of its implementing class ClassPathXmlApplicationContext. The configuration information is stored in the xml file, spring1.xml, will be used by the container to get the bean details.

Example that uses ApplicationContext (Contd.).

spring1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="fig" class="com.wipro.Rectangle"  />
</beans>
```

*** Point to remember : Place this xml file under the src folder(right click the src folder and create this file). This is because you are using ClassPathXmlApplicationContext and this object looks for resources starting from src folder(which happens to be the classpath folder).*

Execute this code and observe the result. It should print the following output on the console screen :

The area of Rectangle is 200.0

Example that uses ApplicationContext (Contd.).

Point to ponder :

When we execute the code given in the previous page, the application displays the following output on the console :

The area of Rectangle is 200.0

What if, you want to display the area of Triangle? In other words, how do you make the AreaClass execute the area method from Triangle class?

Bean Definition Configuration File

All the basic definitions of the Bean classes along with the Configuration Information, their relationships with other Bean objects can be defined in the ***Xml Configuration File***.

Sample Configuration file :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
        <bean id="fig" class="com.wipro.Rectangle"  />
</beans>
```

Sample Client code :

```
ApplicationContext context = new ClassPathXmlApplicationContext("spring1.xml");
Figure fig = (Figure)context.getBean("fig");
```

When the container reads the above configuration file, it understands that it needs to instantiate an object of the type Rectangle, which is placed in a package called com.wipro and make this object available whenever the client invokes getBean method with an argument “fig”. You can observe from the configuration file that “fig” represents the bean id.

Bean Life Cycle

The Bean object defined in the Xml Configuration File undergoes a ***Standard Lifecycle Mechanism.***

Lifecycle interfaces like **InitializingBean** and **DisposableBean** are available to enhance/modify the lifecycle.

The **InitializingBean interface** has a single method called **afterPropertiesSet(),** which will be called immediately after all the property values that have been defined in the Xml Configuration file are set.

The **DisposableBean** has a single method called **destroy(),** which will be called during the shut down of the Bean Container.

Source: http://www.javabeat.net/articles

Example demonstrating Bean Life Cycle

Example code illustrating the usage of 'Life Cycle Interfaces'

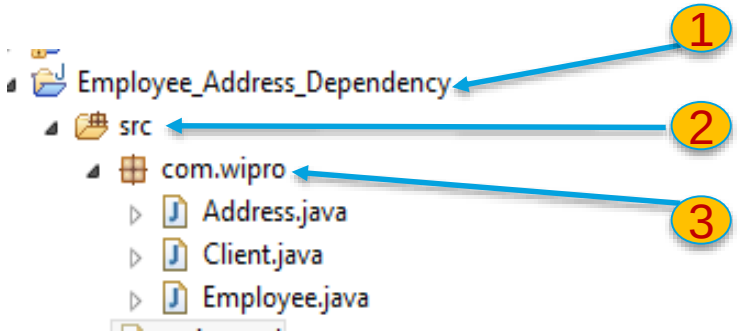
```
import org.springframework.beans.factory.*;
public class Employee implements InitializingBean, DisposableBean {
    private String name;
    private String id;
    public void afterPropertiesSet() throws Exception {
        System.out.println("Employee->afterPropertiesSet() method
called");
    }
    public void destroy() throws Exception {
        System.out.println("Employee->destroy() method called");
    }
}
// Setter/Getter methods
```

Only the bean class details are given here. The participants are expected to execute this example by adding the class containing main method and the xml file.

Source: http://www.javabeat.net/articles

Quiz

Given the folder structure



And the following client code :

```
public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("xyz/spring.xml");
        ……..
}
```

Where will you place the file spring.xml?

1. Create a folder xyz within the folder Employee_Address_Dependency(project folder), which is represented by arrow 1 and create spring.xml within it(xyz)
2. Create a folder xyz within folder src, represented by arrow 2, and create spring.xml within this folder(xyz).
3. Create a folder xyz within the package com.wipro, represented by arrow 3 and place spring.xml within it
4. None of the above

Quiz

1. The class “ClassPathResource” implements which of the following interfaces ?

a. ClassPath
b. Resource
c. BeanFactory
d. ApplicationContext

2. The class “ClassPathXmlApplicationContext“ implements which of the following interfaces?

a. ClassPathXml
b. ClassPathContext
c. Resource
d. ApplicationContext

Thank You