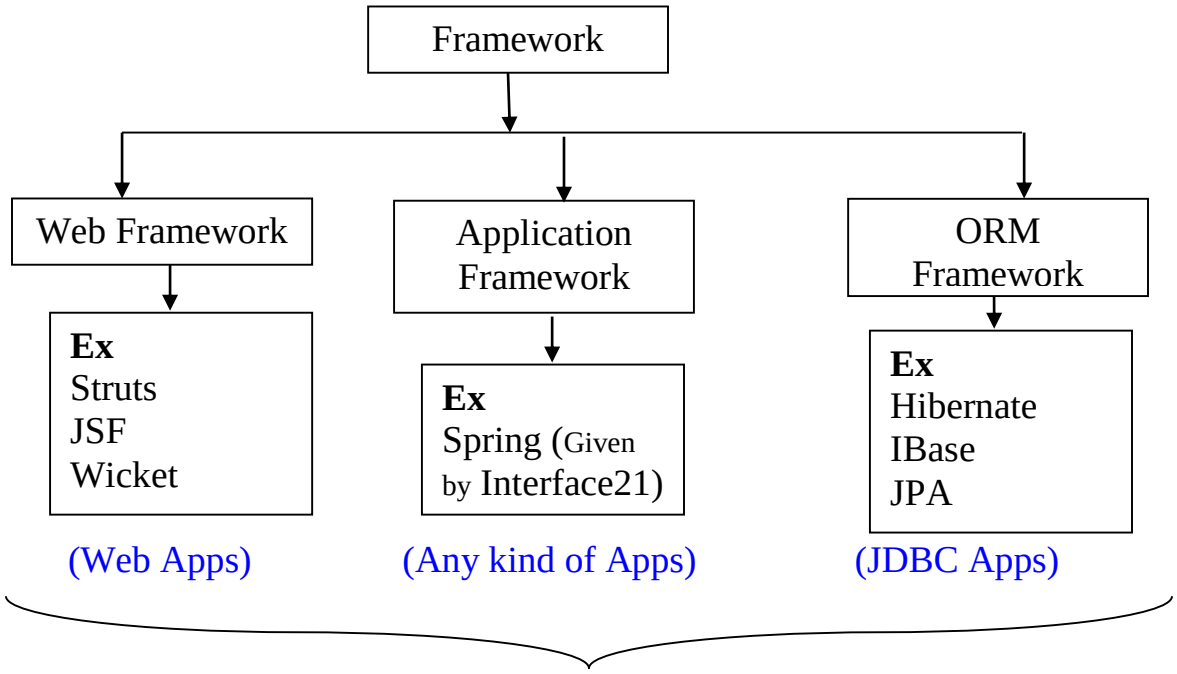
Framework (F/W)

1. Framework is a Software.
2. It is an abstraction layer of existing technology (like JSE, JEE).
3. Framework is not part of JSE and JEE, JME but Framework S/W is using JSE and JEE APIs.
4. F/W is providing a predefined support to developer in application development.
5. F/W is a semi-finished application.
6. F/W is a special kind of S/W which comes in the form of set of jar files and it makes an applications development in less time.



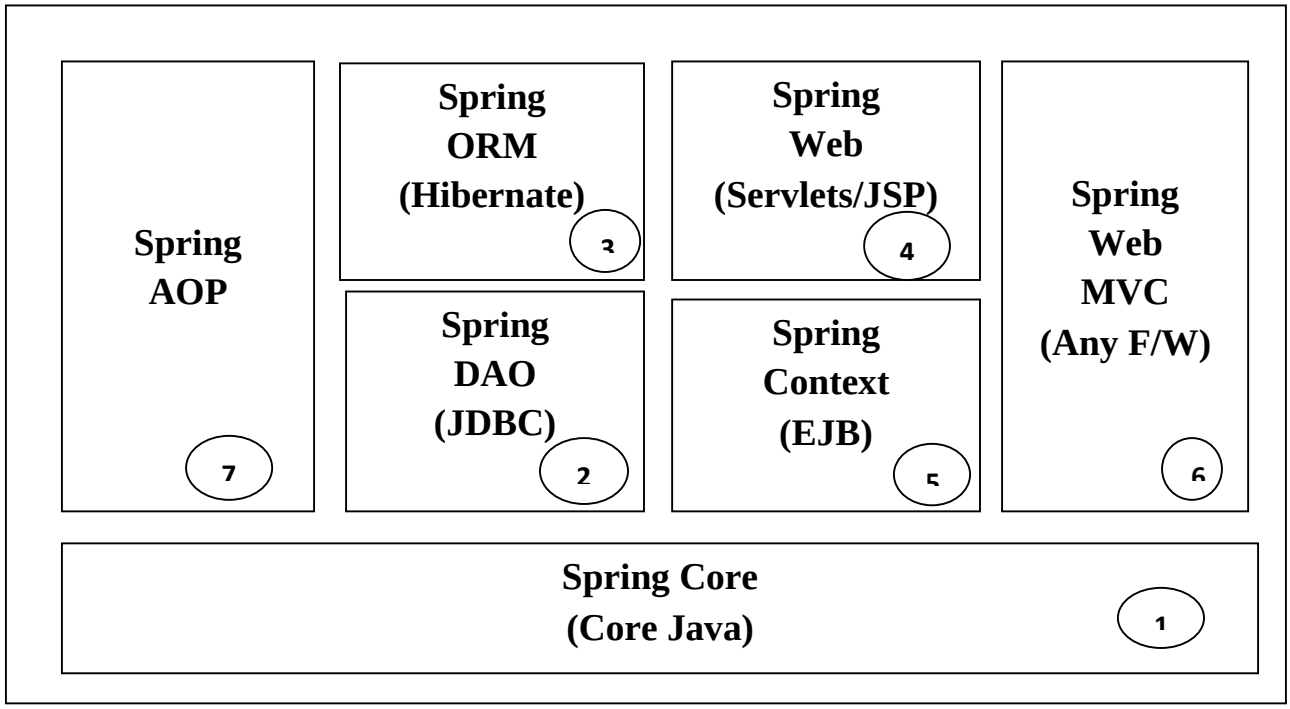
Divided into two Parts (Invasive & Non-Invasive)

Note: Container is a Software that manage application life cycle from birth to death.

Spring Versions:

1.x → 7 Modules
2.x → 6 Modules
3.x → 20 Modules (6 Major)

Architecture of Spring 1.x



Types of Frameworks

There are two types of F/Ws-

1. Invasive F/Ws

2. Non-Invasive F/Ws

1. Invasive F/Ws:

It forces the developers to extend or to implement their predefined class and interfaces (which are given by F/W S/W).

Ex – Struts 1.x

2. <u>Non-Invasive F/Ws:</u>

It does not force the developers to extend or to implement their predefined class and interfaces (which are given by F/W S/W). It is also light weight compare to other F/Ws, because it doesn't use third party container.

Ex – Spring
Struts 2.x
Hibernate

Struts is Web F/W.

By using web F/W only it is possible to develop web applications.

Hibernate is ORM (Object Relational Mapping) F/W.

By using ORM F/W only it is possible to implements persistence login (Data Access Layer).

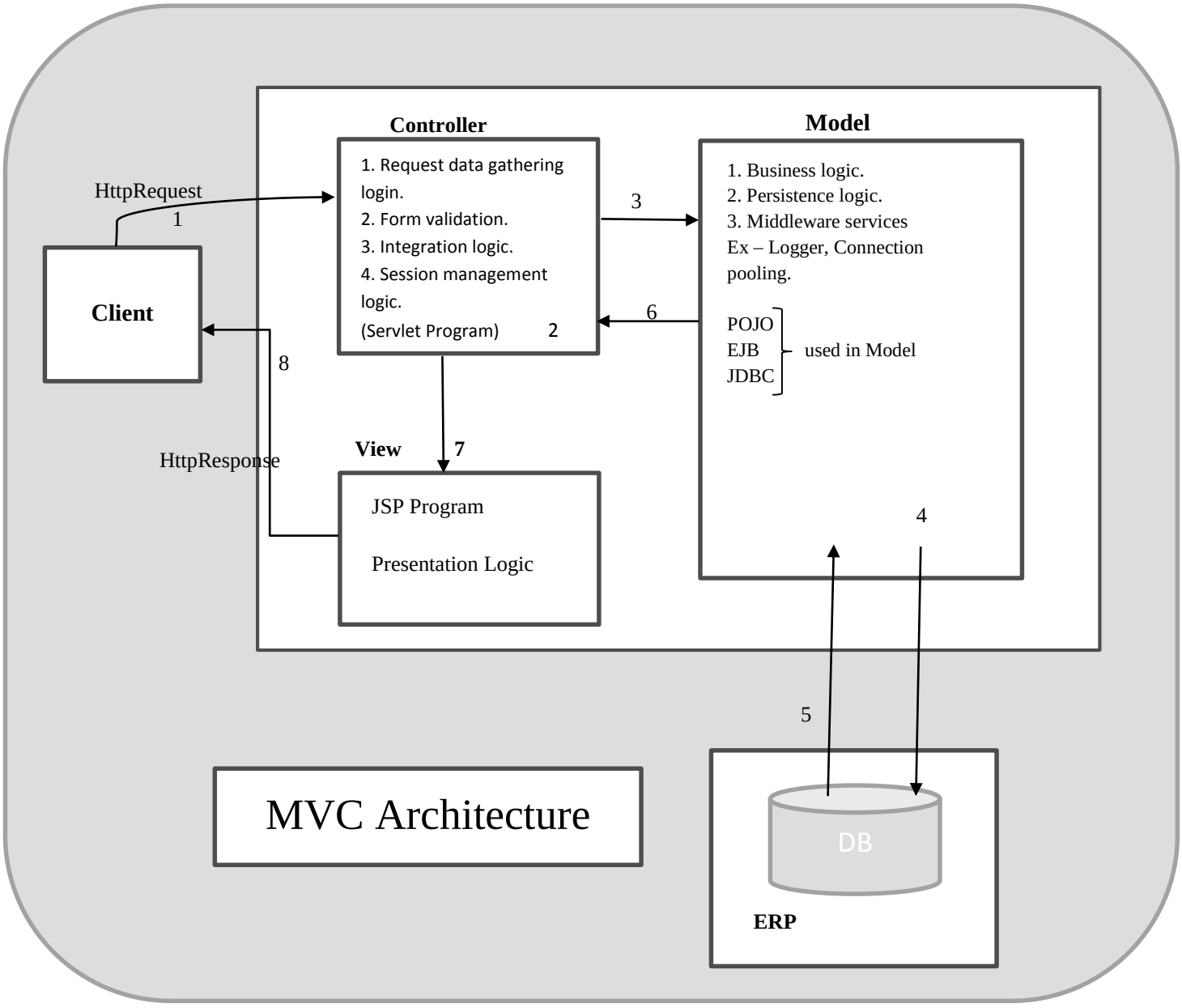
Spring is an application F/W.

By using Application F/W, it is possible to develop any Java Application i.e. Standalone, Enterprise (Web, Distributed apps).

It is also light weight because it uses its own containers.

Spring is an open source.

All types or logics are possible to develop using Spring.



Spring:

- Spring is an application framework.
- It is open source and light weight.
- It is Non-invasive F/W.
- Spring F/W will allow **POJO** and **POJI** model programming.
- By using Spring F/W it is possible to develop all kinds of applications and also possible to implements all type of logics.

Type: Spring is an Application F/W.

Versions:
1.x (jdk 1.3+)
2.x (compatible with jdk 1.5+)
3.x (compatible with jdk 1.6+)

Vendor: Interface21
Creator: Mr. Rod Jhonson

Q. Why spring is called as light-weight?
Ans.

- Spring F/W allows POJO and POJI model programming.
- To execute most of the spring application it doesn't need the heavy weight web/application servers.
- The basic jdk S/W along with Spring jar files is enough to execute the Spring applications.
- Spring Supplies its own containers in the form of predefined classes.

Note:

Servlets, JSP, EJB containers are heavy weight containers.

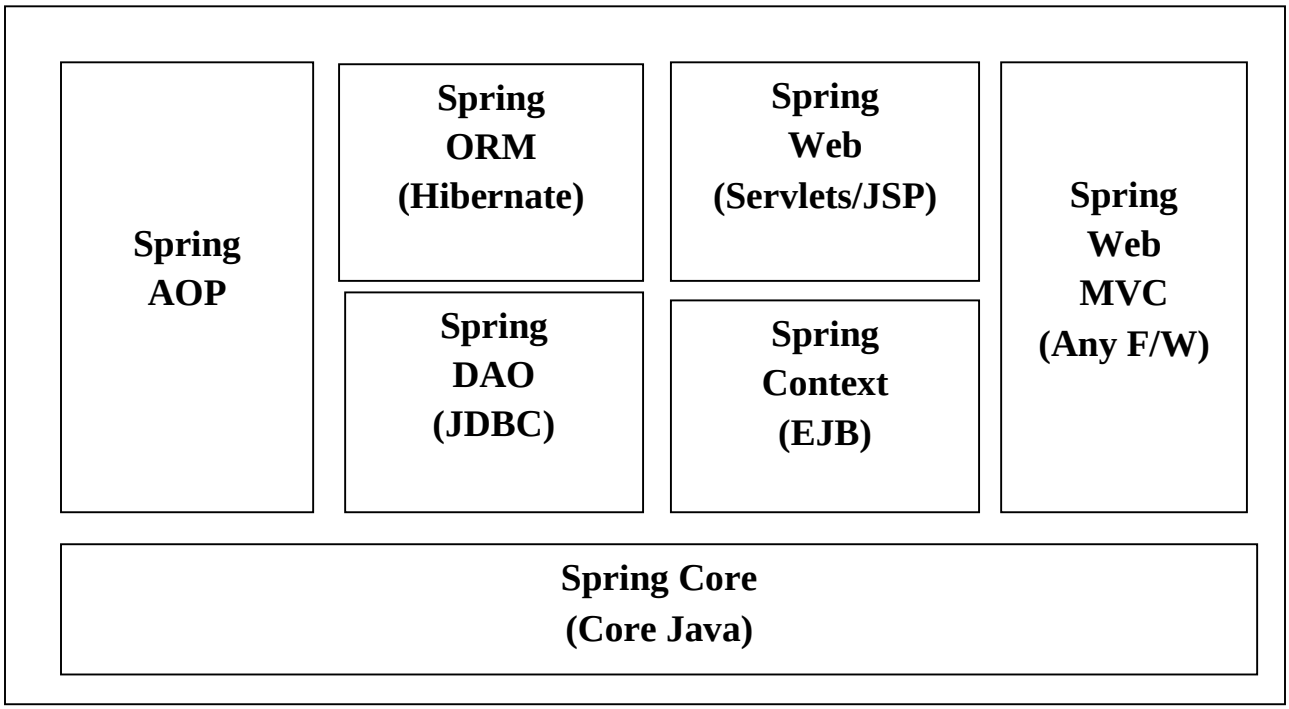
Spring Architecture:

- Spring is well-organized architecture consisting of different modules.
- Spring 1.x contains 7 modules.
- Spring 2.x contains 6 modules.
- Spring 3.x contains 20 modules, but they are grouped into 6 major categories.

Modules of Spring F/W:

Spring 1.x :

1. Core Module
2. DAO Module
3. ORM Module
4. Context Module
5. AOP Module
6. Web Module
7. Web MVC Module



1. Core Module:

- Spring core module is base module for all the modules.
- This module concentrates on major principals of **IOC (Inversion of Control).**
- We can call **IOC is the heart of Spring Application.**
- The core module concentrating on **Dependency Injection (initialization).**

2. DOA Module:

- It provides abstraction layer on JDBC API.
- This model allows us to do the Database operations by using template classes.
- It is used to build the Data Access Layer of an Enterprise Application.

3. ORM Module:

- It provides an abstraction layer on ORM F/W Software like Hibernate, Ibatis, JPA, Toplink ete.
- It is used to build Data Access Layer of an Enterprise Application.

4. Context Module:

- It provides an abstraction layer on JEE Technologies like Java Mail API, JMS (Java Message Services), EJB, RMI, Web Services etc…
- It is used to implement Business Login (Service Layer) of an Enterprise Application.

5. AOP Module: [Aspect Oriented Programming]

- Aspect means Middleware Services.
- Aspect Oriented Programming is a Technique or a Methodology.
- It provides the suggestions to configure the Middleware Services.

- To divide the Middleware Services from Business Logic (BLogic) and to execute the Middleware Services along with the BLogic.

6. Web Module:

- Spring Web module is given to make the Spring Applications to communicate with other web F/Ws like Struts, JSF, Wicket etc…

7. Web MVC Module:

- Spring Web MVC is given to develop Model 2 architecture applications by using Spring own controller classes and custom action tags.

Spring 2.x:

- In Spring 2.x there are 6 modules…
  1. Core Module
  2. JDBC/DAO Module
  3. AOP Module
  4. JEE/Application Context Module
  5. ORM Module
  6. Spring Web Module

Note:

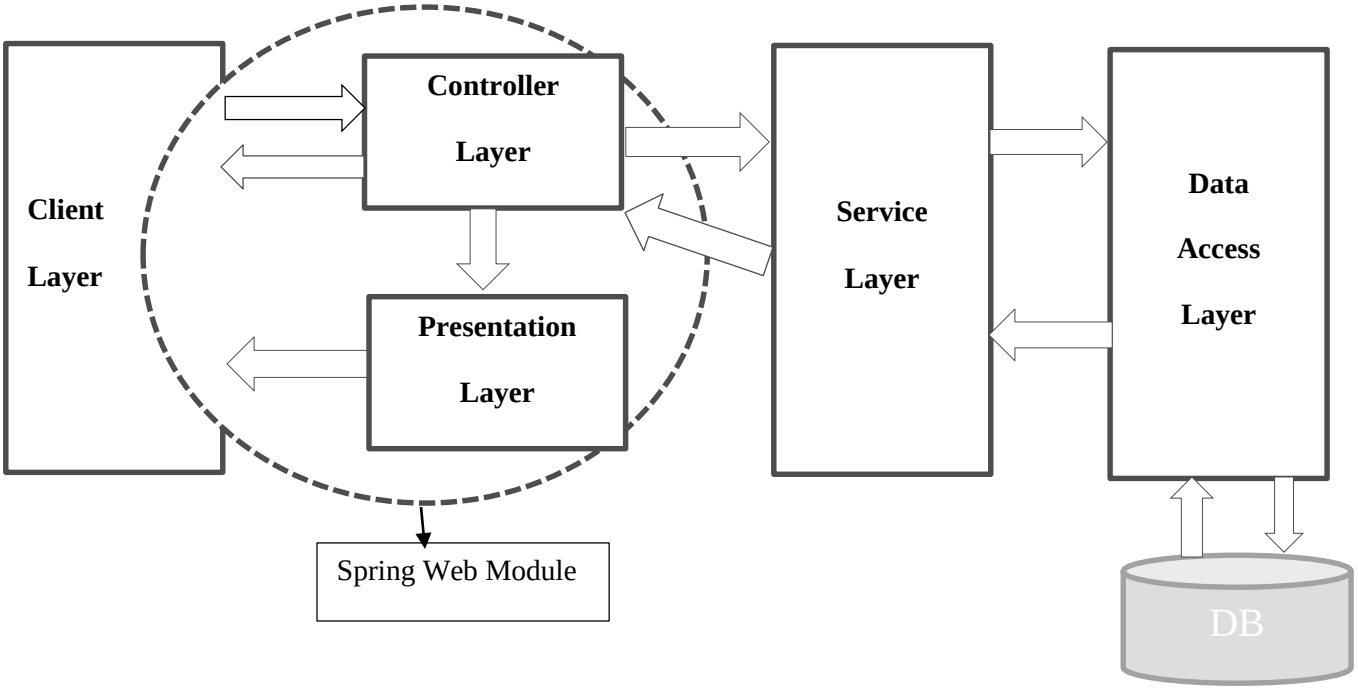
Spring 2.x JEE Module mean →

Spring 1.x Context Module + Misc information

Spring 2.x Web Module means →

Spring 1.x Web Module + Spring 1.x Web MVC Module

Architecture of Enterprise Application:



1. Business Logic:

The main logic of the application is called Business Logic (BLogic).
Ex – in Banking application perform Deposit, Withdraw logic, whatever logic we use that logic is called BLogic.

2. Controlling Logic:

Controlling logic is used to control entire application. It is act as a Traffic Police.
It is also used to get the communication b/w View Layer and Model Layer resources.

3. Presentation Logic:

Presentation logic views user interface to end user to supply inputs to the application and view the results of the application.

4. Persistence Logic:

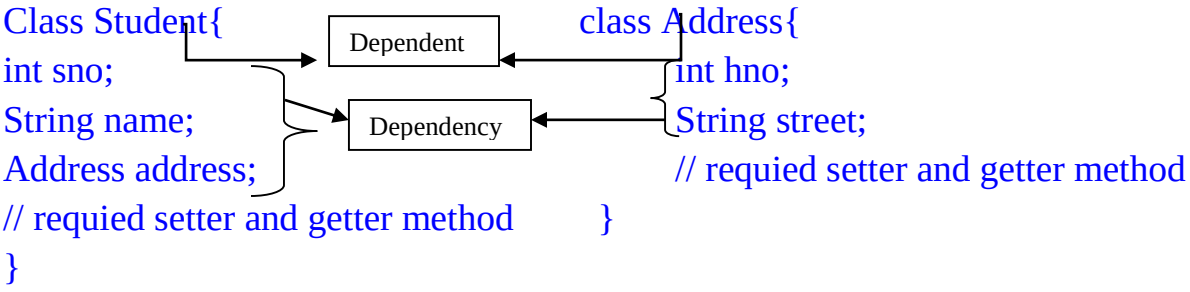
The logic that is used to communicate with persistence store (File/DB) is called persistence logic.

1. Spring Core Module:

- Spring Core Module is the base module of Spring Framework.
- It is used to develop Standalone Applications.
- Spring Core Module gives **BeanFactory Container.** It is designed for supporting **Dependency Injection** or **IOC.**
- The container supports bean life cycle operations.
- Bean object managed by Spring Container.
- Spring Core Module mainly concentrating on **Dependency Injection (DI).**

Dependency Injection: [DI]

- The process of injecting/pushing/initialization to dependencies into an object (Dependent Object) is known as **Dependency Injection.**



Advantages of Dependency Injection:

- Dependency will be reduced i.e. it will provide loose coupling.
  Coupling ➔ the degree of the dependency b/w the components is called coupling.
- If the degree of **dependency is high** b/w the components then that component are called **Tight-coupled.**
- If the degree of **dependency is low** b/w the components then that component are called **Loose-coupled.**
- With loose coupling enhancement of an application is easy or simple.
- It provides dynamic polymorphism.

IOC: [Inversion of Control]

- IOC states that as a programmer we should focus on use of Object.
- IOC will maintain the creation, distribution and dependency satisfaction of Objects.
- It is a design principal
- It is an external entity of DI i.e. injecting the dependency into dependent object.
- Spring container performs the Dependency Injection with IOC.

Container:

- It is Software or Software application or a Java class.
  Ex – Servlet Container takes care of the whole life cycle of Servlet program (i.e. from birth to death).
  Spring container takes care of the whole life cycle of spring bean (i.e. from birth to death).

Resources to Develop a Spring Application:

To develop Spring Application the below 4 resources are requied…

1. Interface [optional]
2. bean class
3. Spring configuration File
4. Client Application (main class)

1. Interface:

- It is an optional resource of Spring Application.
- It contains declaration of Business Methods.
- It can be **POJO** or **Non-POJI.**

POJO: [Plain Old Java Object]

- When a java class is taken as resources of certain java technology based application and if that class not extended and not implemented from a predefined class and predefined interfaces of that Technology specific API is known as **POJO** class.
- Ex - When a java class is taken as resources of Spring based application and if that class not extended and not implemented from a predefined class and predefined interfaces of Spring F/W is known as **POJO** class.

```
class Demo{
        // Demo is POJO
}
class Sample extends Demo{
        // Sample is POJO
}
class Demo extends Action{
        // Demo is not a POJO, because Action is Struts class
}
class Demo extends AbstractController{
        // Demo is not POJO, because AbstractController is Spring class.
}
class Demo implements Serializable{
        // Demo is a POJO, because this interface is in java.io package
}
class Sample extends Demo{
        // Sample is POJO
}
```

POJI: [Plain Old Java Interface]

- When a java interface is taken as a resource of certain java Technology based application and if that interface is not extending any that respective Technology specific API interfaces, then it is called **POJI.**

Note:

bean class is always a POJO, but POJO class may or may not be a bean.

2. bean:

- It can be any java class that's support instantiation (object creation).
- It implements Spring interface (optional).
- It can be POJO or Non-POJO.
- It contains constructors, setxxx() methods and required Business method to support **DI.**
- Spring bean need not be a java bean but java bean can act as a Spring bean.

```
class Test{
    public static void main(String[] args){
            Class cls = Class.forName("java.lang.Thread");
            System.out.println(cls.getClass());
            Object obj = cls.newInstance();
            Thread t = (Thread)obj;
            System.out.println(t.getClass());
    }
}
```

3. Spring Configuration (cfg) File:

- Spring configuration contains bean definition or configuration and other information.
- It is any file name .xml file can be taken as Spring configuration file.
- There is no default name.
- The configuration file bean definition related to DI.
- Configuration file contains configurations related to bean life cycle operations.
- In configuration file <beans> element is a root element.
- Every xml file contains either **DTD (Document Type Definition) or XSD (XML Schema Definition).**
- DTD & XSD are techniques used to frame the rules that are required to construct an xml document.
- Spring configuration can be developed either based on DTD rules or based on XSD rules.

Note:

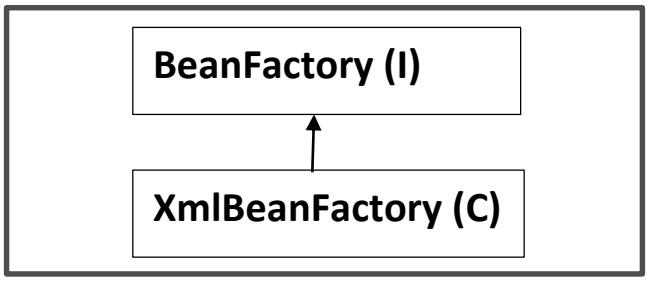
One important feature of Spring core container is that is does not force the programmers/ users. To have any one format of configurations, but most of the F/Ws which are designed to have xml based or .property file based configurations, but Spring at present allows us to supply the configuration either as an xml format or .property file format or programmatically using the Spring API.

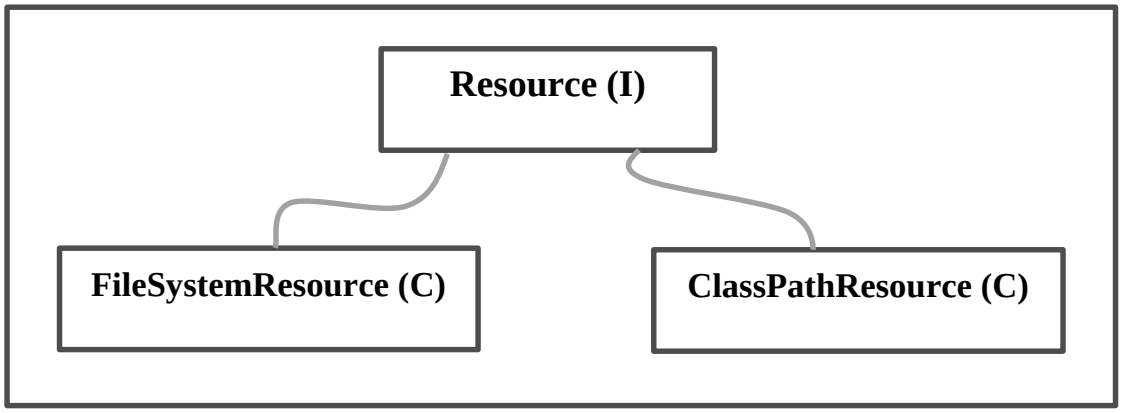
4. Client Application: [Main Class]

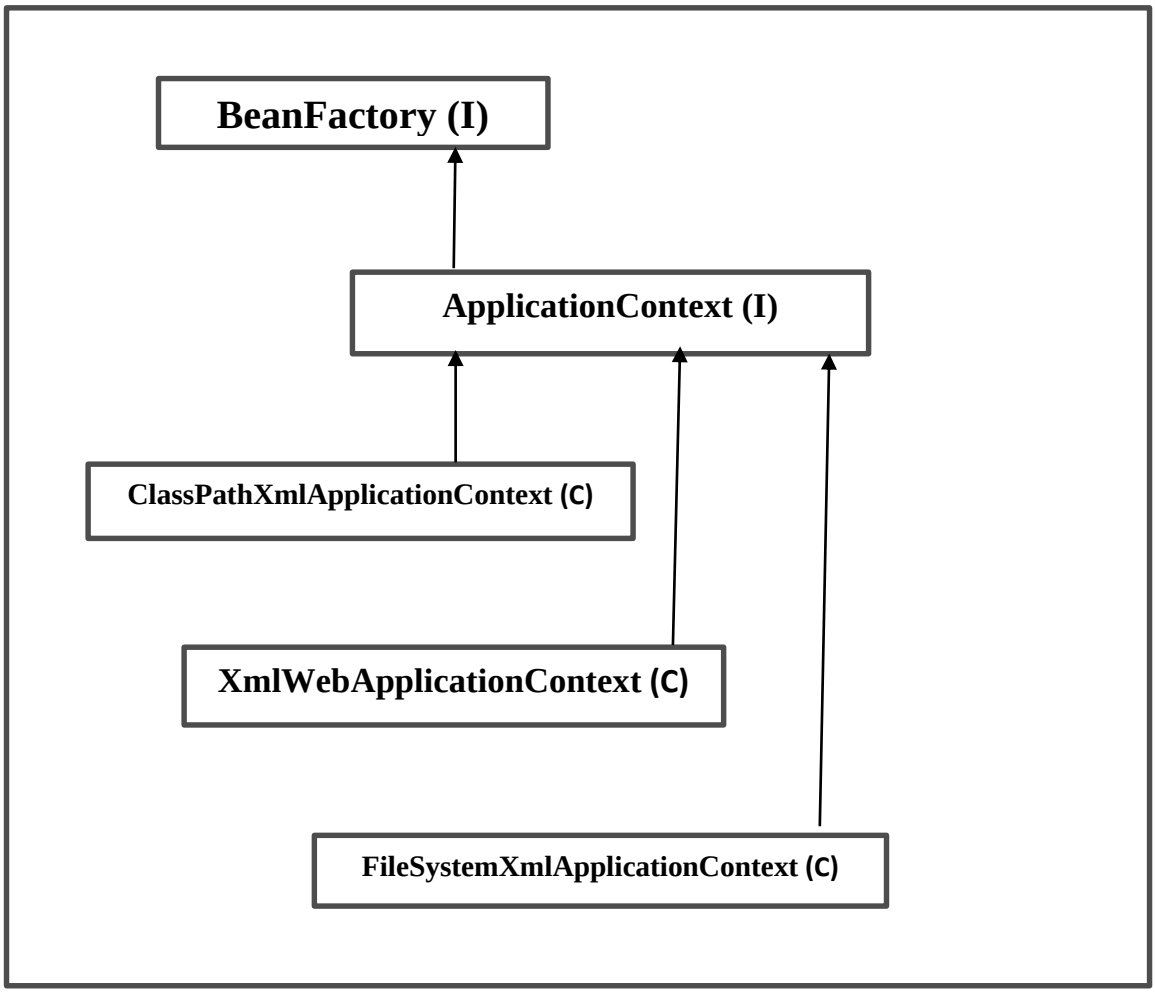
- The Spring core container can be instantiated by creating an object of any one of the BeanFactory or ApplicatonContext interface.

BeanFactory:

- BeanFactory is an interface present in **org.springframework.beans.factory** package.
- Its implementation class name is **XmlBeanFactory.**
- XmlBeanFactory class is present in **org.springframework.beans.factory.xml** package.
- To create XmlBeanFactory object, we can use the following constructor **XmlBeanFactory(Resource r).**







Note:

- BeanFactory has given by Spring Core Module.
- ApplicationContext container has given by Spring Context Module.

Key Points on BeanFactory Container:

- BeanFactory container instantiate and manages all the bean objects.
- BeanFactory container creates the bean object whenever we call **getBean()** method.
- getBean() is present in BeanFactory interface.

Public Object getBean(String beanName)

- In the client application first we can create container object.
- Get the container object reference from the container by calling getBean() method.
- Call required business method as per your application requirement.
- Spring container supports 3 types of Injections…

  1. Setter Injection
  2. Constructor Injection
  3. Interface Injection (only for know about Spring beans and other beans managed by container)

1. Setter Injection:

The Spring container performs the injection by executing Spring **bean class** setter methods then it is called Setter Injection.

2. Constructor Injection:

If Spring container performs the injection by executing Spring bean class **constructor** then it is called constructor injection.

Q. How to implement setter injection?

Ans.

Step 1:

In Spring bean class declare the required dependency variables.

Ex –

```
package com.nareshit;
public class Student{
    private int studentId;
    public void setStudentId(int studentId){
        this.studentId = studentId;
    }
    public getStudentId(){
        return studentId;
    }
```

}

Step 2:

Use the <property> tag in bean configuration file and supply the value.

Ex –

<beans>
<bean id = “std” class = “com.nareshit.Student”>
<property name = “student” value = “1001”/>
Or
<property name = “student” value = “1001”>
</property>
Or
<property name = “student”>
<value>1001<value/>
</property>
</bean>
</beans>

- In Spring configuration file <beans> tag is a root tag. It encloses all the Spring bean definitions.
- <bean> tag described a Spring bean.
- <bean> tag id attribute holds the object name.
- Class attribute is used to specify bean class name.
- <property> tag is used to describe one setter method property and which allows only one argument.

  Ex –

  <property name = “student” value = “1001”/>

  |_**stdObject.setStudentId(100);**

With the <property> tag container calls the corresponding setter method to perform the injection.

- <property> tag support 3 attributes…

I. name attribute:

- This attribute is used to specify bean object property name.

- It follows a java beans convention like if property name is Uname then corresponding setter method name is **setUname(String uname).**

Note:

In Spring primitive types, wrapper object types, String types are called Simple types.

II. value attribute:

- This attribute describe content of Simple type properties in String representation, which is converted into corresponding argument type internally by using the java beans property Editors.

III. ref attribute:

- This attribute describes a reference of a bean i.e. if the property is object type to pass the object reference we can use **ref attribute.**

Application Structure:

F:\Spring Programs\FirstApp
|
|-------Hello.java
|-------myBeans.xml
|-------Student.java
|-------Test.java
|-------com
        |-------nareshit
                |------Hello.class
                |------Student.class
                |------client
                        |-----Test.class

Hello.java

```
package com.nareshit;

public interface Hello{
        public String sayHello();
}
```

Student.java

```
package com.nareshit;

public class Student implements Hello{
        private String studentName;
        public void setStudentName( String studentName){
                this.studentName = studentName;
        }
        public String getStudentName(){
                return studentName;
        }
        public String sayHello(){
                Return "Hello "+studentName+" Welcome";
        }
}
```

myBean.xml

<! DOCTYPE……Copy from below file
org\springframework\beans\factory\xml\spring-beans-2.0 DTD File

To get this file, extract below file
org.springframework.beans-3.0.1.RELEASE.jar
">
<beans>
<bean>
<property name = "student" value= "Raj">

```
</property>
</bean>
</beans>
```

|__The bean object is created internally by using newInstance().

```
Class cls = Class.forName(beanClassName);
Object obj = cls.newInstance();
```

Client Application
Test.java

```
package com.nareshit.client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xm.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import com.nareshit.Student;
public class Test{
        public static void main(String[] args){
        Resource r = new FileSystemResource("f://FirstSpringApp/myBeans.xml");
        BeanFactory container = new XmlBeanFactory(r);
        // get Bean Object from container for this we can call the following method
        // from BanFactory
        // public Object getBean(String beanName)
        Object obj = container.getBean("std");
        Student student = (Student)obj;          // downcasting
        String mst = student.sayHello();
        System.out.println(msg);
        }
}
```

Note:

Every Framework S/W contains 2 types of jar file…

1. Main jar files
2. Dependent jar files

- ⇨ Main jar files are required at the time of compilation and as well as execution.
- ⇨ Dependent jar files are required only at the time of application execution.

***.** To compile and execute above application we need to set the following jar files into the classpath -

Spring-core-3.2.0.M2.jar → Main jar file
Spring-beans.3.2.0.M2.jar → Main jar file
Commons-logging.jar → Dependent jar file

To download above jar files-

http://mvnrepository.com/atifact/org.springframework/spring-core/3.0.0.RELEASE

http://mvnrepository.com/atifact/org.springframework/spring-bean/2.0.0.RELEASE

Note:

- Spring container performs the **Dependency Injection** only on those spring bean object, if that are created by spring container itself. That means Spring container can't perform **DI** on Spring bean class object that are created by developer manually.
- When the BeanFactory container is activated it reads and verifies the entries of spring configuration file.
- Spring container internally uses **SAS parser** of an XML to read and process spring configuration file entries.
- XML parser is a S/W application that can validate XML document against DTD/Schema rules and read, process the XML document.

FileSystemResource:

- It is an implemented class of Resource interface.
  Ex –
  Resource r = new FileSystemResource("f://spring/FirstApp/com/nareshit/xml/myBean.xml");
- FileSystemResource is taking complete system path (absolute path), so in the future, if we change the location of the project then again we need to change the location of spring configuration file in the program configuration file in the program (in the client application).

ClassPathResource:

- It is an implemented class of Resource interface.
- It is present in org.springframework.core.io package
  Ex –
  Resource r = new ClassPathResource ("com/nareshit/xml/myBean.xml");
- The ClassPathResource container reads the configuration file from the classpath, even if you change the location of the project in the system still our project will work without modifications, because it reads the configuration file from classpath. So it is not advisable to use FileSystemResource. It is always recommended to use ClassPathResource.

Application Structure:

SpringExample1

```
|------src
|       |-------com.nareshit.bean
|       |       |------Employee.java
        |-------com.nareshit.client
        |       |------Test.java
        |-------com.nareshit.config
        |       |------myBean.xml
        |-------com.nareshit.dao
```

```
|       |------EmployeeDao.java
|       |------EmployeeDaoImpl.java
|-------com.nareshit.service
        |------EmployeeService.java
        |------EmploeeServiceImpl.java
[-]JRE System Library
```

//Employee.java

```
package com.nareshit.bean;
public class Employee{
    private int empNo;
    private String Name;
    private double Salary;
//required setter and getter methods
    Public int getEmpNo(){
        return empNo;
    }
}
```

//EmployeeService.java

```
package com.nareshit.service;
import com.nareshit.bean.Employee;
public interface EmployeeService{
    int createEmloyee(Employee emp);
}
```

//EmployeeServiceImpl.java

```
import com.nareshit.bean.employee;
public class EmployeeService{
    private EmployeeDao empDao;
    public void setEmpDao (EmployeeDao empDao){
        this.empDao = empDao;
    }
    public int createEmployee (Employee emp){
        int count = empDao.saveEmployee(emp);
        return count;
```

```
        }
}

//EmployeeDao
import com.nareshit.bean.employee;
public interface EmployeeDao{
        int saveEmployee (Employee emp);
}

//EmployeeDaOImpl
import java.sql.connection;
public class EmployeeDaoImpl implements EmployeeDao{
        private connection con;
        public EmployeeDaoImpl(){
                try{
                        Class.forName("oracle.jdbc.driver.OracleDriver");
                        String url = "jdbc:oracle:thin:@localhot:1521:XE";
                        String user = "system";
                        String pass = "rajtomar";
                        con = DriverManager.getConnection(url,user,pass);
                }
                catch(ClassNotFoundException e){
                        e.printStackTrace();
                }
                catch(SQLException e){
                        e.printStackTrace();
                }
        }
        public int saveEmployee (Employee emp){
                int count = 0;
                String sql = "insert into employee values(?,?,?)";
                try{
                        PreparedStatement pst = con.prepareStatement(sql);
                        pst.setInt(1,emp.getEmpNo());
                        pst.setString(2,emp.getName());
```

```
                pst.setDouble(3,emp.getSalary());
                count = pst.executeUpdate();
            }
            catch(SQLException e){
                e.printStackTrace();
            }
            return count;
        }
}
```

//myBean.xml

```
<beans>
<bean id = "empServiceObj"
class = "com.nareshit.service.EmployeeServiceImpl">
<property name = "empDao" ref = "empDaoObj"/>
</bean>
</beans>
```

//Test.java

```
public Test{
    public static void main(String[] args){
    Resource r = new ClassPathResource("com/nareshit/config/myBean.xml");
        BeanFactory factory = XmlBeanFactory(r);
        Objec obj = factory.getBean("empServiceObj");
        EmployeeService eservice = (Employee)obj;
        Employee emp = new Employee();
        emp.setEmpNo(1001);
        emp.setName("Raj");
        emp.setSalary(30000);
        int count = eservice.createEmployee(emp);
        if(count>0){
            System.out.println("Employee Created");
        }
        else{
            System.out.println("Employee not Created");
```

```
        }
    }
}
```

Constructor Injection:

Step 1:

Create the bean class with the required properties (dependencies) and with the required constructors.

Step 2:

Make the user of **<constructor-arg>** tag inside the <bean> tag. If the constructor parameter is simple type to pass the value, we can use **<value>** attribute. If the constructor parameter is Object type, use **<ref>** attribute.

Ex –

ConstructorInjectionExample1

```
|
|------com.nareshit.bean
|       |------Employee.java
|------com.nareshit.client
|       |------Test.java
|------com.nareshit.xml
        |------myBean.xml
```

//Employee.java

```
public class Employee{
    private int empNo;
    private String empName;
    public Employee(int empNo, String empName){
        this.empNo = empNo;
        this.empName = empName;
    }
    Public void getEmployeeDetails(){
        System.out.println("EmpNo: "+empNo);
        System.out.println("EmpName: "+empName);
```

```
        }
}
```

//myBean.xml

```
<beans>
<bean id = “emp” class = “com.nareshit.bean.Employee”>
<constructor-arg value = “1001”/>
<constructor-arg value = “Raj”/>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] agrs){
        Resource r = new ClassPathResource(“com/nareshit/xml/myBean.xml”);
        BeanFactory factory = new XmlBeanFactory(r);
        Employee emp = (Employee)factory.getBean(“emp”);
        emp.getEmployeeDetails();
        }
}
```

Constructor Ambiguity Problem:

Ex –

//Employee.java

```
public class Employee{
        private int eno;
        private String name;
        private double sal;
        private String design;
        public Employee(int eno, String name){
                this.eno = eno;
```

```
            this.name = name;
        }


        public Employee(double sal, String degis){
            this.sal= sal;
            this.desig= desig;
        }

        public void getEmployeeDetails(){
            System.out.println("Eno: "+eno);
            System.out.println("Name: "+name);
            System.out.println("Salary: "+Sal);
            System.out.println("Designation: "+desig);

        }

}
```

//myBean.xml

```
<beans>
<bean id = "emp" class = "com.nareshit.bean.Employee">
<constructor-arg value = "1001"/>
<constructor-arg value = "Raj"/>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] agrs){
        Resource r = new ClassPathResource("com/nareshit/xml/myBean.xml");
        BeanFactory factory = new XmlBeanFactory(r);
        Employee emp = (Employee)factory.getBean(emp);
        emp.getEmployeeDetails();
        }
```

}

- In the above example program we can't expect exact output, because Employee is having two constructors with 2 arguments. In configuration <constructor-arg> takes the value in String format and if convert into corresponding type. So it may assign to Employee of eno, name or sal, desig.
- To avoid this ambiguity problem we can work with following two attributes-
  Index attribute
  Type attribute
  ⇨ Index is used to specify the index of the constructor argument. It takes from zero.
  ⇨ Type is used to specify the type of the constructor argument for type we must we fully qualified names.
  ⇨ To avoid the ambiguity problem we can above configuration file as follows-

```
<beans>
<bean id = "emp" class = "Employee">
<constructor-arg value = "1001" type = "int", index = "0"/>
<constructor-arg value = "Raj" type = "java.lang.String", index = "1"/>
</bean>
</beans>
```

Note:

- If all the bean properties are configured for **setter injection**, then spring container create the spring bean class object using **zero argument constructor.**

- If all the bean properties are configured for **constructor injection**, then spring container create the spring bean class object using **parameterized argument constructor.**
- If same bean property are configured for **constructor injection** and same bean property for **setter injection**, then spring container create the spring bean class object using **parameterized constructor.**

Q. What happen if same bean property is configured for setter and constructor injection with different values?
Ans.
<bean id = "std" class = "Student">
<property name = "name" "raj"/>
<contructor-arg value = "raja"/>

name = ~~raj~~
raja

Since setter method executes after constructor execution, so the value passed through setter injection will override the value injected by constructor injection but bean object well be created by using parameterized constructor.

Q. How to inject a List based collection in a spring bean?
Ans.
Step 1: With the spring bean class (dependent class) declare List type variable as a dependency.
Ex – class College{
List<Student> listOfStudent;
}

Step 2: Make the use of collection configuration element in bean configuration file i.e. we use **<list>** tag inside <property> or <constructor> tag.

ListTypeBasedInjectionExample

```
|------src
|       |-------com.nareshit.bean
|       |       |--------College.java
|       |       |--------Student.java
|       |-------com.nareshit.client
|               |--------Test.java
|------myBean.xml
```

//Student.java

```
package com.narshit.bean;
public class Student{
        private  int sid;
        private String name;
        public String toString(){
                return "Student id : "+sid+"\n"+"Name : "+name;
        }
        //required getter and setter methods
}
```

//College.java

```
package com.nareshit.bean;
import java.util.List;
public class College{
        private String collegeName;
        private List<Student> listOfStudetn;
        public College(Sting collegeName){
                this.collegeName = collegeName;
        }
        public void setListOfStudent(List<Student> listOfStudent){
                this.listOfStudent = listOfStudent;
        }
        public String getCollegeName(){
                return collegeName;
```

```
        }
        public List<Student> getListofStudent(){
                return listOfStudent;
        }
}
```

//myBean.xml

```
<beans>
<bean id = "studentObj1" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1001"/>
<property name = "name" value = "raj"/>
</bean>
<bean id = "studentObj2" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1002"/>
<property name = "name" value = "tomar"/>
</bean>
<bean id = "studentObj3" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1003"/>
<property name = "name" value = "raja"/>
</bean>
<bean id = "collegeObj" class = "com.nareshit.bean.College">
<constructor-arg value = "NIT">
</constructor-arg>
<property name = "listOfStudent">
<list>
<ref bean = "studentObj1">
<ref bean = "studentObj2">
<ref bean = "studentObj3">
</list>
</property>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                Resource r = new ClassPathResource("myBean.xml");
                BeanFactory f = new XmlBeanFactory(r);
                College c = (College)f.getBean("collegeObj);
                System.out.println(c.getCollegeName());
                List<Student> listOfStudent;
                c.listOfStudent();
                for(Student.listOfStudent){
                        System.out.println(Student);
                }
        }
}
```

Q. What are the difference between **ref bean and ref local?**

Ans.

- A local attribute looks for the given id based dependent spring bean the current configuration file only.

  Ex –

  <bean id = "s1" class = "Student">
  <property name = "address"/>
  <ref local = addressObj> // => local search in current Object
  </property>
  </bean>

- **ref bean** attribute searches for the dependent bean in the current configuration file if it is not available in current configuration file, then it search parent configuration file.

  Ex –

  <bean id = "studentObje" class = "Student">
  <property name = "address">
  <ref bean = "addressObj"/>

</property>
</bean>

<property name = “address” ref = “addressObj”/>

In the above example <ref = “addressObj”/> is as <ref bean = “addressObj”/>

<bean id = “std” class = “Student”>
<property name = “address” ref = “addressObj”/>

Is equal to

<bean id = “std” class = “Student”>
<property name = “address”>
<ref bean = “addressObj”/>

Note:

In <constructor-arg> re attribute is there but is not local attribute.
<constructor-agr ref = “addressObj”>

Or

<contructor-arg>
<ref bean = “addressObj”/>
</constructor-arg>

Q. How to inject a Set based collection in Spring bean?

Ans.

Step 1: In the spring bean class (dependent class) declare a Set type variables as dependency variable.

Ex – class college{

Set<Student> setOfStudent;

}

Step 2: Make the user of collection configuration element in the bean configuration file i.e. we can use **<set>** tag.

Ex –

```
<property name = “setOfStudents”>
<set>
<ref bean = “studentObj1”/>
<ref bean = “studentObj2”/>
<ref bean = “studentObj3”/>
</set>
</property>
```

SetTypeBasedInjectionExample

```
|------src
|       |-------com.nareshit.bean
|       |       |--------College.java
|       |       |--------Student.java
|       |-------com.nareshit.client
|               |--------Test.java
|------myBean.xml
```

//Student.java

```
package com.narshit.bean;
public class Student{
        private  int sid;
        private String name;
        public String toString(){
                return “Student id : ”+sid+”\n”+”Name : ”+name;
        }
        public int hashCode(){
                return sid;
        }
        public Boolean equals(Object obj){
                if(obj instanceof Student){
                        Student std = (Student)obj;
                        return this.sid = std.sid && this.name = equals(std.name);
                }
                else{
                        return false
```

```
            }
        }
        //required getter and setter methods
}
```

//College.java

```
package com.nareshit.bean;
import java.util.Set;
public class College{
        private String collegeName;
        private Set<Student> setOfStudents;

        // required setter and getter Methods
}
```

//myBean.xml

```
<beans>
<bean id = "studentObj1" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1001"/>
<property name = "name" value = "raj"/>
</bean>
<bean id = "studentObj2" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1002"/>
<property name = "name" value = "tomar"/>
</bean>
<bean id = "studentObj3" class = "com.nareshit.bean.Student">
<property name = "sid" value = "1003"/>
<property name = "name" value = "raja"/>
</bean>
<bean id = "collegeObj" class = "com.nareshit.bean.College">
<constructor-arg value = "NIT">
</constructor-arg>
<property name = "setOfStudents">
<set>
```

```
<ref bean = "studentObj1"/>
<ref bean = "studentObj2"/>
<ref bean = "studentObj3"/>
</set>
</property>
</bean>
</beans>
```

//Test.java (like List)

```
public class Test{
        public static void main(String[] args){
                Resource r = new ClassPathResource("myBean.xml");
                BeanFactory f = new XmlBeanFactory(r);
                College c = (College)f.getBean("collegeObj);
                System.out.println(c.getCollegeName());
                List<Student> listOfStudent;
                c.listOfStudent();
                for(Student.listOfStudent){
                        System.out.println(Student);
                }
        }
}
```

- Set object is used to represent a group of objects as single units.
- Set collection objects are not allowed duplicates. If we are adding any duplicate then no compile time error and no run time error, simply add() method returns false.

equals() Method:

- It is given to compare two objects of a class for their equality.
- It compares the objects based **implementation logic.**
- In **Object class equals()** is implemented to compare the two objects with the references.

- Inside the Object class equals() method implementation –

```
public class Object{
    public boolean equals(Object obj){
            return this == obj;
    }
}
```

- If you want to compare two objects of a class with the state, we can override equals() method in your user-defined class.

(***) Contract between equals() & hashCode() Methods:

- If equals() method returns **true** –
  If and only if two objects hashcode must be same.

- If equals() method returns **false** –
  That two objects hashcode may or may not be same.
- To satisfy the above contract we can override the hashCode() method, whenever are overriding equals() method.

Note:

Generally whenever we are working with the HashTable Data Stucture (Set, HashMap, HashTable) to avoid the duplicate objects (by comparing that objects with state) we are overriding equals() and hashCode() methods.

Note:

"==" operator compares the two objects with the references only, but equals() method compare the two objects based on Implementation Logic.

To know about Package ➔[Javap java.util.Map$Entry]

Q. How to Inject a Map based collection into spring bean?
Ans.
Step 1: In the spring bean class declare **java.util.Map** type variable as a dependency variable.
Step 2: Make the use of <map> tag in spring configuration file.

- Map is used to define a group of objects as a key and value pairs in Map each key and value as one entry.
- Map is act as root Interface. it is not a child interface of Collection.

Ex –
<map>
<entry key = "key1" value = "1"/>
<entry key = "key2" value ref ="studentobj1"/>

```
<entry key = “key3”>
<bean id = “studenobj2” class = “Student”>
<property name = “rollno” value = “1001”/>
<property name = “name” value = “rama”/>
</bean>
</entry>
…….
…….
</map>
```

Application Structure

MapTypeBasedInjectionExample1

```
|------src
|       |-------com.nareshit.bean
|       |       |--------Question.java
|       |
|       |-------com.nareshit.client
|               |--------Test.java
|------myBean.xml
|------log4j.properties
```

//Question.java

```
Import java.util.Map;
public class Question{
        private int questioned;
        private String question;
        private Map<String, String> answers;

        // required setters and getters Methods
}
```

```
// myBean.xml
<DOC TYPE….>
<beans>
<bean id = “questionObj” class  = “nareshit.com.bean.Question”>
<property name – “questionedId” value = “1001”/>
<property name – “questioned” value = “What is Java?”/>
<property name – “answers”>
<map>
<entry key  = “java is an Object Oriented” value = “posted by rama”/>
<entry key  = “java is an platform independent” value = “posted by srinu”/>
<entry key  = “java is an programming language” value = “posted by rama”/>
</map>
</property>
</bean>
</beans>

//Test.java

import java.

public class Test{
        private final static Logger logger = Logger.getLogger(Test.class);
        public static void main(String[] agrs){
                Resource r = new ClassPathResource(”myBean.xml”);
                BeanFactory factory = new XmlBeanFactory(r);
                Logger.infor(“BeanFactory container created”);
                Question question = (Question)factory.getBean(“questionObj”);
                System.out.println(“Qustion Id: “+question.getQuestion());
                System.out.println(“Qustion : “+question.getQuestion());
                Map<String, String> answers = question.getAnswers());
                logger.info (“No of Entries in Map : “+answers.size());
                Set<Entry<String, String>> set = answers.entrySet();
                Iterator<Entry<String, String>> itr = set.iterator();
                while(itr.hasNext()){
                        Entry<String, String> entry = itr.next();
```

```
                String value = entry.getValue();
                System.out.println("Answers : "+key" : "+value);
            }
        }
}
```

//log4j.properties

```
#log4j property file to write output on log
#file and show on console as well

log4j.rootLogger = INFO, file, stdout
#Direct log messages to a log file

Log4j.appender.file =
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File = E:\\logginFile.org
log4j.appender.file.MaxFileSize = 1MB
log4j.appender.file.MaxBackupIndex = 1
log4j.appender.file.layout = org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversioinPattern = %d{yyyy-MM-dd HH:mm:ss} % -
5p %c{1}:%L - %m%n

#Direct log messages to console

log4j.appender.stdout =
org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target = System.out
log4j.appender.stdout.layout =
org.apache.log4j.PatterLayout
log4j.appender.stdout.layour.ConversionPatter = % {yyyy-MM-dd HH:mm:ss} %-
5p %c {1}: %L - %m%n
```

Log4j:

- It is a flexible library as a open source project from apache.
- System.out.println is a basic debugging Statement only to print the statement to console, but log4j can be used to print the statement to console, file, Database or Email or remote server.
- Main components in log4j are -

1. Appender:

This is class which does the actual logging (printing statements). It can be console appender, File appender, JDBC appender and so on…

2. Layout:

This is nothing but how the log message is formatted. This format is very similar to C Languages printf() function.
Ex – "Hello I am training java tools"

3. ConversionPattern

It is similar to C language format specifiers.

Application Structure

MapTypeBasedInjectionExample2

```
|------src
|       |-------com.nareshit.bean
|       |       |--------Question.java
|       |       |--------Answer.java
|       |       |--------User.java
|       |-------com.nareshit.client
|               |--------Test.java
|------myBean.xml
|------log4j.properties
```

//Question.java

```
public class Question{
        private int questioned;
        private String question;
        private Map<Answer, User> answers;

        //required setter and getter methods
}
```

//Answer.java

```
import java.util.*;
public class Answer{
        private int answered;
        private String answer;
        private Date postedDate;

        public int getAnswerId(){
                return answerId;
        }
}
```

//User.java

```
public class User{
        private int userId;
        private String userName;
        private String email;
// required setter and getter
}
```

//myBean.xml

```
<DTD>
<beans>
<bean id = “user1” class = “com.nareshit.bean.User”>
<property name = “userId” value = “1001”/>
<property name = “userName” value = “Ramu”/>
<property name = “email” value = “ramu@gmail.com”/>
</bean>
<bean id = “user2” class = “com.nareshit.bean.User”>
<property name = “userId” value = “1002”/>
<property name = “userName” value = “3sha”/>
<property name = “email” value = “3sha@gmail.com”/>
</bean>

<bean id = “ans1” class = “com.nareshit.bean.Answer”>
<property name = “answerId” value = “2001”/>
<property name = “answer” value = “Java is Programming Language”/>
<property name = “postedDate” value = “12/12/2012”/>
</bean>
<bean id = “ans2” class = “com.nareshit.bean.Answer”>
<property name = “answerId” value = “2002”/>
<property name = “answer” value = “Java is Object Oriented”/>
<property name = “postedDate” value = “11/12/2013”/>
</bean>
<bean id = “ans3” class = “com.nareshit.bean.Answer”>
<property name = “answerId” value = “2003”/>
<property name = “answer” value = “Java is Simple”/>
<property name = “postedDate” value = “3/03/2013”/>
</bean>

<bean id = “questionObj” class = “com.nareshit.bean.Question”>
<property name = “questionId” value = “3001”/>
<property name = “question” value = “What is Java?”/>
<property name = “answers”>
<map>
```

```
<entry key-ref = “ans1” value-ref “user1”/>
<entry key-ref = “ans2” value-ref “user2”/>
<entry key-ref = “ans3” value-ref “user1”/>
</map>
</property>
</bean>
</beans>
```

//Test.java

```
public class Test{
        private static Logger logger = Logger.getLogger(Test.class);
        public static void main(String[] args){
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource
                (“myBean.xml”));
        logger.info(“BeanFactory container created”);
        Question q = (Question)factory.getBean(“questionObj”);
        System.out.println(q.get(Question));
        Map<Answer,user> answers = q.getAnswers();
        logger.info(“No of Elements in Map: +answers.size());
        Set<Entry<Answer,User>> set = answers.entrySet();
        Iterator<Entry<Answer,User>> itr = set.iterator();
        while(itr.hasNext()){
                Entry<Anwer,User> entry = itr.next();
                Answer answer = entry.getKey();
                User user = entry.getValue();
                Sysetm.out.println(“Answer : “);
                System.out.println(answer.getAnswer()+”Posted  Date
                :”+answer.getPostedDate());
                System.out.println(“”Posted By : “);
                System.out.println(“Name : “+user.getUserName()+”Email:
                “+user.getEmail());
                logger.info(“Answers Posted by “+user.getUserName());
                logger.info(“Mail id “+user.getEmail());
```

```
            }
        }
}
```

//log4j.properties (with html layout)

```
log4j.rootLogger = INFO, file
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File = springMapMapBasedInjection.html
log4j.appender.file.layout = org.apache.log4j.HTMLLayout
log4j.appender.file.append = true
```

BeanFactory Container internal work flow to create a Bean Object

1. Loading the XML bean definitions from resource [myBean.xml].
2. Just before loading the bean definition first it will found the beans DTD.
3. After reading the DTD (Spring Container will use SAX parser to read the spring configuration file) bean definitions will load.
4. Whenever we are calling the getBean() method container will load the bean class byte code and creates that bean objects by using newInstance() method.

```
//loads the bean class byte code
Class cls = Class.forName("beanClassName");
                                    // the required constructor only it is trying to
//it will get the constructors      // convert.
Constructor[] cons = cls.getDeclaredConstructors();
if(cons[0].modifier("isPrivateAccessability"){
        cons.setAccessible (true"); →//there may be chance of rising
}                                           //   securityException
Object obj1 = cons.newInstance();
```

Factory Method:

- **If any java class method is having the capability of for constructing and returning its own object reference that method is called factory method.**
- There are two types of factory methods in java-

  1. static factory method
  2. non-static factory method or instance factory method

Note:

If any java class contains private constructor then it is not possible to create an object by using new operator and constructor from ***Out Side of the class.***

⇨ static factory method is very useful to create an Object for private constructor classes from Out Side of the class.

- In some situations in spring it is not possible to create the object for spring bean class by using new operator and constructor. In this case to create bean object by using static factory methods.

Note:

We can make the spring container to create the spring bean class object either by using static factory method or by using instance factory method or by using regular constructor.

With static factory method:

- With static factory methods to create spring bean class object we can use **<factory-method>** attribute along with <bean> configuration.
  Ex –

<bean id – "cls" class = "java.lang.Class">
</bean> **// container will rise secutiryException**

<bean id = “cls” class “java.lang.Class” factory-method = “forName”>
<constructor-arg value = “java.lang.thread”/>
</bean>

This code is equal to →
Class cls = Class.forName(“java.lang.Thread”);

- The factory method attribute indicates to container to create bean object by using static factory method.
- ***To specify the argument values of a factory method use <constructor–arg> tag.***
- In this case <constructor-arg> is not constructor injection.

Instance factory method:

- The instance factory method is used to create an object by using an existing object and its data.
  Ex –

FactoryMethodExample1

```
|------src
|       |-------com.nareshit.bean
|       |       |--------Employee.java
|       |       |--------Test.java
|
|------myBean.xml
```

//Employee.java

```
public class Employee{
        public Employee(){
                System.out.println("Employee cons");
        }
        public Eployee getEmployee(){
                System.out.println("Factory Method of Employee");
                return new Employee();
        }
}
```

//myBeans.xml

```
<beans>
<bean id = "emp1" class = "com.nareshit.beans.Employee"/>
<bean id = "emp2" factory-bean = "emp1" factory-method = "getEmployee"/>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource
                ("myBeans.xml"));
                Employee emp1 = (Employee)factory.getBean("emp1");
                System.out.println(emp1.hashCode());
                Employee emp2 = (Employee)factory.getBean("emp2");
                System.out.println(emp2.hashCode());
        }
}
```

- While configuring the spring bean ***"factory-method"*** attribute is places along with ***class attribute*** then spring container uses ***static factory method*** for instantiation.
- While configuring the spring bean if ***factory-method*** attribute is placed along with ***factory-bean*** attribute and without class attribute then spring container uses instance factory method to create spring bean class object.

Connection connection = DriverManager.getConnection(…..);

is equal to below code

```
<beans>

<bean id = "connection" class = "java.sql.DriverManager"
factory-method ="getConnection">
<constructor-arg value = "jdbc:oracle:thin:@localhost:1521:XE"/>
<constructor-arg value = "system"/>
<constructor-arg value = "manager"/>
</beans>
```

(***)java.util.Properties Type Based Injection:

- To perform the injection on ***java.util.Properties*** type dependencies we can use **<*props*>** tag.
- This tag will create java.util.Properties object.
- To add on key and value to Properties object we can use **<*prop*>** tag inside the **<*props*>** tag.

Note:

java.uti.Properties Object is used to maintain the group of String objects as a key and value pairs.

Ex –

```
<props>
<prop key = "username">system</prop>
<prop key = "password">manager</prop>
</props>
```

PropertiesTypeBasedInjectionExample1

```
|
|-----src
|       |-------default package
|       |-------DatabaseConnection.java
|       |-------Test.java
|---muBean.xml
```

//DatabaseConnection

```
public class DatabseConnection{
        private Properties dbProperties;
        // required setter and getter methods
}
```

//myBean.xml

```
<beans>
<bean id = "databaseConnction" class = "DtabaseConnection">
<property name = "dbProperties">
<props>
<prop key = "driverClassName">oracle.jdbc.driver.OracleDriver</prop>
<prop key = "url">jdbc:oracle:thin:@localhost:1521:XE</prop>
<prop key = "username">system</prop>
<prop key = "password">manager</prop>
</props>
</property>
</bean>
</beans>
```

//Test.java

```
public class Test{
        BeanFactory factory = new XmlBeanFactory (new
                ClassPathResource("myBean.xml"));
        DatabaseConnection db =
         (DatabaseConnection)factory.getBean("dataBaseConnection");
        Properties p = db.getDbProerties();
```

```
        Enumeration enum = p.keys();
        while(enum.hasMoreElements()){
                String key = (String)enum.nextElement();
                String value = p.getProperty(key);
                System.out.println(key+" : "+values);
        }
}
```

BanFactory(I)

|

ApplicationContext(I):

|

|----------ClassPathXmlApplication(c)

|----------XmlWebApplicationContext(c)→ [used in web applications]

|----------FileSystemXmlApplicationContext(c)

- ApplicationContext Container is given by spring context module.
- ApplicationContext is an child interface of BeanFactory interface.
- ApplicationContext interface present in org.springframework.context package.

Note:

BeanFactory container ***not supporting to read the data from .properties*** files.

- BeanFactory container not supporting to maintain multiple configuration files in the application.
- BeanFactory container creates the bean object whenever we call getBean() method.
- BeanFactory container is used in standalone applications.

Key Points on ApplicationContext Container:

- ApplicationContext Container generally used in enterprise based applications.
- ApplicationContext container create the spring beans at the time of loading the configuration file i.e. it loads the spring beans configured in the spring configuration file and manages the life cycle of the spring bean as when container starts.
- It won't wait until calling of getBean() method.

ClassPathXmlApplicationContext(C):

- It is an implemented class of ApplicationContext interface.
- It is present is org.springframework.context.support.

Ex –

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

- In the above code container ***reads the configuration*** file from the ***classpath***. So even if we change the location of the project in the system our application will work without the modification, because it reads the configuration file from the classpath not from the system path.
- It is recommended to use ClassPathXmlApplicationContext container (FileSystemXmlApplicationContext is not recommended).

FileSystemXmlApplicationContext:

- It is an implemented class of ApplicationContext interface.
- It is present in org.springframework.context.support package.

Ex –

```
ApplicationContext context = new FileSystemXmlApplicationContext
("F://spring/core/application/5pm/FactoryMethodExample1/src/myBean.xml");
```

- In the above code the container reads the configuration file from the system path (Absolute path).
- If you change the project location in the system then we must be change application code. So it is not recommended to use.

Used jar files in context-

[asm, context, expression]

How to Download Maven and Install:

To download the Maven Software use the following link-
http://maven.apache.org/download.cgi

From above link we can download ***apache-maven-3.2.2-bin.zip***

After download extract that zip file and set the path environment variables for Maven Software.

To set the path environment variables go for system variables we can set –
JAVA_HOME C:\Program Files…….\jdk1.7.0
MAVEN_HOME E:\apache-maven

Provide the path in user variables for maven home
%MAVEN_HOME%\bin

To check Maven working or not
C:\users\USER>mvn install

Spring with Maven Integration:

1. To create the project select - File--> new-->other->Maven Project
2. Click on Next button
3. Again click on Next button
4. Select - org.apache.maven.archetypes maven-archetype-quickstart

5. Click on Next button
6. Enter the ***Group Id*** as Package Name ex – com.nareshit
   Artifact Id ad Project Name ex – SimpleSpringMavenProject
7. Click on Finish.

SpringMavenProject1
|
|------com.nareshit.bean
| |-----Employee.java
|------com.nareshit.client
| |-------Test.java
|-----src/test/java
[-] JRE System Library
[-] Maven Dependencies
|------src
| |----main
| | |-----resources
| | | |-----myBean.xml
| |-----test
|------target
|------pom.xml

//Employee.java

```
public class Employee{
        Private int empno;
        private String name;
// required setter and getter methods
}
```

//myBean.xml

```
<beans>
<bean id = “emp” class = “com.nareshit.bean.Employee”/
<property name = “empno” value = “1001”/>
<property name = “name” value = “rama”/>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                BeanFactory factory = new XmlBeanFactory
                                        (new ClassPathResource ("myBeans.xml"));
                Employee emp = (Employee)factory.getBean("emp");
                System.out.println("emp.getName());
        }
}
```

//pom.xml

```
<modelVersion>4.0.0</modelVersion
<groupId>com.nareshit</groupId>
<artifactId>SpringMavenProject1</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>SpringMavenProject1</name>
<url>http::://maven.apache.org</url>.

<dependencies>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-beans</artifactId>
<version>3.1.1.RELEASE</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>3.1.1.RELEASE</version>
</dependency>
<!--  <dependency>
<groupId>commons-logging</groupId>
<artifactId>commons-logging</artifactId>
<version>1.1.2</version>
</dependency>
```

</dependencies>
</project>

groupId

- The groupId element is supposed to identify a particular project (or) set of libraries within a company or organization.
- Generally we will use the package name as <groupId> name.

artifactId

- It represents the actual name of the project.
- It combines with the groupId and should be uniquely identified the project.

version

- Every project also has a <version> element, which indicates the current version number.
- This number usually refers to major releases (“Hibernate 3.2.4”, “Spring 3.0.1” and so on).

modelVersion

- It indicates POM model version (always 4.0.0).

Auto Wiring (Implicit Injection):

- Auto wiring ***means automatically injecting*** the dependencies (implicit dependency Injection Operation).
- Instead of manually configuring (Explicit Dependency Injection), the injection we can do automatically by using auto wiring.
- To implement the auto wiring we can use ***<autowire>*** attribute in bean tag configuration.

Limitations of auto wiring:

- Auto wiring is possible only on referenced type bean properties that means auto wiring is not possible on simple type, collection framework type properties.
- Auto wiring kills the readability of spring configuration file.

To perform auto wiring for a particular bean we can use any one of the following values-

1. byName
2. byType
3. constructor
4. autodetect

1. Auto wiring byName: (*****)

- In the case of auto wiring byName spring IOC container mainly checks for three conditions. If all these are valid then it inject the values by setter aproach.

    a) **Dependency bean name (in dependent class)**
    b) **Configured bean name (in the configuration file)**
    c) **Setter method name (in the dependency class)**

- If dependency name is abc, then bean configuration should be abc and the setter method name should be setAbc(Abc abc)
- When it finds <autowire = “byname”> for any bean configuration, first spring container check for dependency bean name in the dependent bean, then it will check whether any bean is configured in the spring configuration file with the same name.
- If it finds then it will call corresponding setter method of dependent bean.

AutowiringByNameExample1

```
|-----src/main/java
|       |-----com.nareshit.bean
|       |       |-----Employee.java
|       |----com.nareshit.client
|       |       |-----Test.java
|       |-----com.nareshit.dao
|       |       |-----EmployeeDao.java
|       |       |-----EmployeeDaoImpl.java
|       |------com.nareshit.service
|       |       |-----EmployeeService.java
|               |-----EmployeeServiceImpl.java
|------ []src/test/java
[] JRE System Library
[] Maven Dependencies
[] Referenced Libraries
|----src
|       |----main
|       |       |-----resource
|       |               |-----com
|       |                       |----nareshit
|       |                               |----config
|       |                                       |-----myBeans.xml
|       |---test
|-----target
[] pom.xml
```

//Employee.java

```
public class Employee{
        private int empno;
        private String name;
        private double salary;
// required setter and getter method
}
```

//EmployeeService.java

```
public interface EmployeeService{
        public Employee getEmployee(int empNo);
}
```

//EmployeeServiceImpl.java

```
public class EmployeeServiceImpl implements EmployeeService{
        private EmployeeDao employeeDao;
        public void setEmployeeDao(EmployeeDao employeeDao){
                System.out.println("Injection happenedin EmployeeService by using
                                                setEmployeeDao");
        this.employeeDao = employeeDao;
        }
        public Employee getEmployee(int empNo){
                Employee emp = employeeDao.serchEmployee(empNo);
                return emp;
        }
}
```

//EmployeeDao.java

```
public interface Employee{
        Employee searchEmployee(int empNo);
}
```

```
//EmployeeDaoImpl.java
public class EmployeeDaoImpl implements EmployeeDaoImpl{
        private DataSource dataSource;
        public void setDataSource(DataSource dataSource){
                System.out.println("Injection- happened in EmployeeDao by using
                                setDataSource(-)");
                this.dataSource = dataSource;
        }
        public Employee searchEmployee(int empNo){
                Employee emp = null;
        Connection connection = null;
                try{
                        connection = dataSource.getConnection();
                        String sql = "select * form employee where empno = ?";
                        PreparedStatement pst = connection.prepareStatement(sql);
                        pst.setInt(1,empNo);
                        ResultSet rs = pst.executeQuery();
                        if(rs.next()){
                                emp = new Employee();
                                emp = setEmpNo(rs.getInt(1));
                                emp = setName(rs.getString(2));
                                emp = setSalary(rs.getDouble(3));
                        }// end of if
                }
                catch(SQLExceptioin e){
                        e.printStackTrace();
                }
                finally{
                        try{
                                if(connection ==null){
                                        connectin.close();
                                }
                        }
                        catch(SQLException se){
                                System.out.println(se);
```

```
                }
            }
            return emp;
        }
}
```

//myBean.xml

```
<beans>
<bean id = “employeeService”
class = “com.nareshit.service.EmployeeServiceImpl” autowire = “byName"/>

<bean id = “employeeDao”
class = “com.nareshit.dao.EmployeeDaoImpl” autowire = “byName"/>


<bean id = “dataSource”
class = “org.springframework.jdbc.darasource.DriverManagerDataSource”>
<property name = “driverClassName” value = “jdbc.oracle.driver.OracleDriver”/>
<property name = “url” value = “jdbc:oracle:thin:@localhot:1521:XE”/>
<property name = connectionProperties”>
<props>
<prop key = “user”>system</prop>
<prop key  = “password”>manager</prop>
</props>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] main){
            ApplicationContext context = new ClassPathXmlApplicationContext
                                            (“applicationContext.xml”);
            EmployeeService empService =
                                (Employee)context.getBean(“employeeService”);
            int eid = 2;
```

```
            Employee emp = empService.getEmployee(eid);
            System.out.println("emp.getName()+" "+emp.getSalary());
        }
}
```

2. Auto wiring byType:

- In the case of auto wiring byType spring container mainly check for 3 conditions, if all these are valid, then it inject the values by setter approach.
    a) Dependency beanType
    b) Configured beanType
    c) Setter methodParameterType
- When it finds ***<autowire = "byType">*** for any bean configuratioin, then first it checks for dependency beanType in the dependent bean, then it will check any bean is configured in the spring configuration file with the same ***Type***. If it finds, then it will call corresponding setter methods.

Note:

In this case container does not bother about the setter method name. It only bothers the setter methods parameter types.

AutowiringByTypeExample1

```
|-----src/main/java
|       |-----com.nareshit.client
|       |       |-------App.java
|       |-----com.nareshit.controller
|       |       |-----EmployeeController.java
|       |------com.nareshit.service
|       |       |------EmployeeService.java
|------src/test/java
[] JRE System Library
[] Maven Dependencies
```

```
[] Referenced Libraries
|----src
|       |----main
|       |       |-----resource
|       |               |-----com
|       |                       |----nareshit
|       |                               |----config
|       |                                       |-----myBeans.xml
|       |---test
|-----target
[] pom.xml
```

//EmployeeController.java

```
public EmployeeController{
        private EmployeeService emplyeeService1;
        public void setEmployeeService2(EmployeeService employeeService1){
                System.out.println("Injection happened in EmployeeController by
                                                        using setEmployeeService");
                this.employeeService1 = employeeService1;
        }
        public void employeeControllerMethod(){
                System.out.println("Employee Controller Method");
                employeeService1.employeeServiceMethod();
        }
}
```

```
//EmployeeService.java
public EmployeeSerice{
        private EmployeeDao employeeDao1;
        public void setEmployeeDao3(EmployeeDao employeeDao1){
                System.out.println("Injection happened in EmployeeService by using
                                                                setEmployeeDao3");
                this.employeeDao1 = employeeDao1;
        }
        public void empServiceMethod(){
                System.out.println("Employee Service Method");
                employeeDao1.empDaoMethod();
        }
}
```

//EmployeeDao

```
public class EmployeeDao{
        public void empDaoMethod(){
                System.out.println("Employee Dao Method");
        }
}
```

//myBean.xml

```
<beans>
<bean id = "employeeController"
class = "com.nareshit.controller.EmployeeController" auwire = "byType"/>
<bean id = "employeeService"
class = "com.nareshit.service.EmployeeService" autowire = "byType"/>
<bean id = "employeeDao"
class = "com.nareshit.dao.EmployeeDao" autowire = "byType"/>
</beans>
```

//Test.java

```
public class App{
        public static void main(String[] args){
                String configFile = "com/nareshit/config/myBeans.xml";
                ApplicationContext context = new ClassPathXmlApplicationContext
                                                        (configFile);
                EmployeeController empController = (Employee)context.getBean
                                                        ("employeeController");
                empController.employeeControllerMethod();
        }
}
```

3. Auto wiring Constructor:

- In the case of auto wiring with constructor spring IOC container mainly check for 3 conditions, if all these are valid then it inject the values by constructor approach.
    - Dependency type
    - Configure bean type
    - Constructor parameter type
- When it finds ***<autowire = "constructor>*** for any bean configuration then it first check for dependency bean type in the dependent bean then it will check the spring configuration file whether any bean is configured with the dependency type.
- If it finds it will check for constructor, which will take dependency type as a parameter, then after it will call corresponding constructor.

//EmployeeController.java

```
public class EmployeeController{
        private employeeService employeeService1;
        public EmployeeController(EmployeeService empService2){
                employeeService1 = employeeService2;
        }
}
```

//EmployeeService.java

```
public class EmployeeService{
        private employeeDao employeeDao1;
        public EmployeeService(EmployeeDao empDao2){
                employeeDao1 = employeeDao2;
        }
}
```

//EmployeeDao.java

```
public class EmployeeDao{
}
```

//myBeans.xml

<beans>
<bean id = “employeeController”
class = “com.nareshit.controller.EmployeeController” autowire = “constructor”/>
<bean id = “employeeService”
class = “com.nareshit.service.EmployeeService” autowire = “constructor”/>
<bean id = “employeeDao” class =”com.nareshit.dao.EmployeeDao”/>
</beans>

4. Auto wiring autodetect:

- Auto-detect choose constructor or byType through injection of the bean classes.
- Always auto-detect first gives the preference to constructor.
- If default constructor or zero argument constructor is found then byType (setter method) applied.

<beans>
<bean id = “employeeController”
class = “com.nareshit.controller.EmployeeController” autowire = “autodetect”/>
<bean id = “employeeService”
class = “com.nareshit.service.EmployeeService” autowire = “autodetect”/>
<beans>

Global default-autowiring:

- Instead of defining autowiring attribute for every bean configuration we can set a default-autowire attribute in the <beans> root element to force the all the beans declared within the <bean> root element to apply this rule.
- However this root default mode will be overridden by a bean tag own mode if it is specified.

<beans default-aotowire = “byType”>
<bean id = “employeeController”
class = “com.nareshit.controller.EmployeeController”/>
<bean id = “employeeService”
class = “com.nareshit.service.EmployeeService” autowire = “constructor”/>
<bean id = “employeeDao”
class = “com.nareshit.dao.EmployeeService” autowire = “no”/>
<beans>

Note:

The autowire attribute will allow following 5 values –

1. no → no autowiring
2. byName
3. byType
4. constructor
4. autodetect

Dependency Checking:

- It is used to verify all the dependency of a bean that are configured via injection are injected or not.
- To implement dependency checking to a spring bean we make use of ***“dependency-check”*** attribute of ***<bean> tag***.
- This attribute take any of the following four values-
    - None
    - Simple
    - All
    - Object

None:

It won’t check whether the dependency injected or not.

Simple:

It checks all simple type (primitive type, wrapper type) dependencies injected or not.

Object:

It checks all the object type dependencies injected or not.

All:

It checks both simple and object type dependencies injected or not.

- If we are using dependency checking by mistake if we forget to set any value any simple or object type, then container raise an exception like – ***"UnSatisfiedDependencyException"***.

Note:

By default the dependency check value is none.

DependencyCheckingExample

```
|----src
        |-----default
                |------Address.java
                |------Student.java
                |------Test.java
```

//Address.java

```
public class Address{
        private String city;
//required setter and getters
}
```

//Student.java

```
public class Student{
        private int studentId;
        private String name;
        private Address address;
```

```
//required setters and getters
}
```

//myBean.xml

```
<xsd
Xsi:schemaLocation = "http://www.springframework.org/schema/beans/spring-beans-2.5 xsd>

<beans>
<bean id = "student" class = "Student" dependency-check = "all"
 autowire ="byType>
<property name = "name" value = "Rama">
</property>
</bean>
<bean id = "address class = "Address">
<property name = "city" value = "hyd"/>
<bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                BeanFactory f = new XmlBF (new CPR("myBeans.xml"));
        Student s = (Student)f.getBean("student");
                System.out.println(s.getName()+":"+s.getStudentId());
        }
}
```

- If we are executing the above application we will get the following exception –

"org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'student' defined in class path resource [myBeans.xml]: Unsatisfied dependency expressed through bean property 'studentId': Set this property value or disable dependency checking for this bean.

Note:

Global dependency checking:

Explicitly using of dependency-check attribute for every bean is increasing the burden so to overcome this problem we can set ***default-dependency-check*** attribute in the beans root element to force all the beans configured within the root elements to apply this rule.
However this root default mode will be overridden by a bean's own mode if it specified.

Ex –
<beans default-dependency-check = "all"
xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:context=http://www.springframework.org/schema/context
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

@Autowired Annotation:

- It is present in org.springframework.bean.factory.annotation package.
- It is used to autowire a bean on the setter method, constructor and field.

Source code of @autowired annotation

```
package org.springframework.bean.factory.annotation;
@Retention (RetentionPolicy.RUNTIME)
@Target ({ElementType.CONSTRUCTOR, ElementType.FIELD,
 ElementType.METHOD})
```

```
public @interface Autowired{
        boolean required() default true;
}
```

- To give the intimation to the spring container about a @Autowired Annotation configuration we can use ***<context:annotation-config/>*** in the spring configuration file.

SpringAnnotationAutowiredExample

```
|------src
|       |-----com.nareshit.client
|       |       |----Test.java
|       |-------com.nareshit.dao
|       |       |------EmployeeDAO.java
|       |-------com.nareshit.service
|               |-------EmployeeeSeervice.java
|------myBeans.xml
```

//EmployeeSerivice.java

```
public class EmployeeService{
        private EmployeeDAO employeeDao;
        @Autowired
        public void setEmployeeDao(EmployeeDAO employeeDao){
                System.out.println("Injection Happened in EmployeeService in using
                                                        setEmployeeDao");
                this.employeeDao = employeeDao;
        }
        public String toString(){
                return "This is Employee Service : Hello" +employeeDao;
        }
}
```

//EmployeeDAO.java

```
publi class EmployeeDAO{
        public String toString(){
                return "EmployeeDAO";
        }
}
```

//myBean.xml

```
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:context=http://www.springframework.org/schema/context
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
<context:annotation-config/>
<bean id = "employeeService" class = "com.nareshit.EmployeeService">
</bean>
<bean id = "employeeDao" class = "com.nareshit.EmployeeDAO">
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                ApplicationContext context = new ClassPathXmlApplicationContext
                                                        ("myBeans.xml");
        EmployeeService eservice = (EmployeeService)context.getBean
                                                        ("employeeService");
        System.out.println(eservice);
        }
```

}

Note:

Dependency Checking:

By default @Autowired annotation will perform the dependency checking to make sure the property has been wired properly, when spring container can't find a matching bean to wire, it will throw an Exception.

We can disable the dependency checking feature by setting ***required*** attribute of @Autowired to false.

The default value of require attribute is true;

Ex –

```
public class Employeeservice{
    private EmployeeDAO employeeDao;
    @Autowired (required = false)  →// if required value is true dependency
                                    //checking is enabled.
    public vod setEmployeeDao(EmployeeDAO employeeDao){
        System.out.println("Injection Happened in EmployeeService by using
                                                    setEmployeeDao");
        this.employeeDao = employeeDao;
    }
}
```

@Qualifier Annotation:

- It is used to control which bean should be autowire on a field (dependency) i.e. @Qualified is used to autowire a particular bean. For example bean configuration file with two similar employeeDao beans.

```
<bean id = "employeeDao1" class = "com.nareshit.dao.EmployeeDAO">
<property name = "empName" value = "Rama"/>
</bean>
```

<bean id = “employeeDao2” class = “com.nareshit.dao.EmployeeDAO”>
<property name = “empName” value = “Krishna”/>
</bean>

- In this case to autowire a particular a bean we can use @Qualifier annotation.

//EmployeeService.java

```
public class EmployeeService{
        private EmployeeDAO employeeDao;
        @Autowired
        @Qualifier (“employeeDao2”)
        public void setEmployeeDao (EmployeeDAO employeeDao){
                this.employeeDao = employeeDao;
        }
        public String toString(){
                return “This is Employee Service : Hello “+employeeDao;
        }
}
```

Spring bean Scopes:

Singleton Java Class:

- The java class that allows to creates only one object for JVM underlying container or run time environment creates only one object for class then only the class is called singleton.
- If the class allows to create multiple objects, then that class is not called singleton java class.
- When a programmer try to create multiple objects for a class, if the java class allows to create only one object, then that class is called as Singleton.

src

```
|------com.nareshit
        |------Car.java
        |------Main.java
```

//Car.java

```
class Car{
        private static Car car = new Car();
        private Car(){
        }
        public static Car getCar(){
                return car;
        }
}
```

//Main.java

```
class Main{
        public static void main(String[] args){
                Car c1 = car.getCar();
                Car c2 = car.getCar();
                Car c3 = car.getCar();
                System.out.println(c1.hashCode());
                System.out.println(c2.hashCode());
                System.out.println(c3.hashCode());
        }
}
```

Here getCar() method allows return the existed Car object reference

Spring Singleton:

- The process of creating only one object for container is called spring singleton.
- We can make the spring container to create the spring bean class object in one of the five following scopes-
  1. Singleton → it is a default value of scope attribute.
  2. Prototype
  3. Request
  4. Session
  5. Global session

1. Singleton:

Spring container creates only one object for spring bean but it never makes spring bean as singleton java class i.e. when scope attribute is singleton the spring container creates and returns one object for spring bean class for multiple factory.getBean() method call. The default value of scope attribute is singleton.

SpringSingletonScopeExample

```
|------src
        |-------com.nareshit.bean
                |-----Employee.java
        |-------com.nareshit.client
                |------Test.java
        |-------com.nareshit.config
                |-------myBeans.xml
```

//Employee.java

```
class Employee{
        private int empNo;
        private String empName;
        private double salary;
// required setter and getter
}
```

//myBeans.xml

```
<beans>
<bean id = “emp” class = “com.nareshit.bean.Employee”
scope = “singleton”/>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                ApplicationContext context =
```

```
                new ClassPathXmlApplicationContext("myBeans.xml");
            Employee emp1 = (Employee)context.getBean("emp");
            emp1.setEmpNo(111);
            emp1.setEmpName("Rama");
            emp1.setSalary(3000);
            System.out.println("emp1: "+emp1);
            Employee emp2 = (Employee)context.getBean("emp");
            System.out.println("emp2: "+emp2);
        }
}
```

Ex –

```
<bean id = "emp1" class = "com.nareshit.bean.Employee" scope = "singleton"/>
<bean id = "emp2" class = "com.nareshit.bean.Employee" scope = "singleton"/>
```

Here two instance will created for Employee class, here bean (scope = "singleton") never make the spring bean class as a singleton java class.

Note:

Singleton beans in spring and singleton design pattern both are different.

1. Scope = Prototype:

If the Spring bean scope = prototype spring container creates and returns multiple objects for spring bean class for multiple factory.getBean() method call.

Ex- in spring configuration file

```
<bean id = "emp1" class = "com.nareshit.bean.Employee" scope = "prototyoe"/>
```

In client Application

```
Employee emp1 = (Employee)context.getBean("emp1");
emp1.setEmpNo(102);
emp1.setEmpName("rama");
emp1.setSalary(3000);
Employee emp2 = (Employee)context.getBean("emp2");
System.out.println(emp1);        //102 rama 3000
```

System.out.println(emp2); // 0 null 0

A new Object is created because scope prototype.

Spring Bean Life Cycle:

- Spring IOC container manages the life cycle of a bean i.e. instantiation to destruction
- Spring bean object has 4 life cycle state.

1. Instantiation
2. Initialization
3. Ready to use
4. Destruction

- Spring framework provides a support of the bean to listen for its life cycle events performed by the IOC container.
- In many real world components have to perform certain type of initialization tasks before they are ready to use such task as opening a file, opening a network or database connection, allocating memory and so on and also they have to perform the corresponding destruction tasks at the end of their life cycle. So we have to need to customize Bean initialization and destruction in the spring IOC container to listen the life cycle events and also to perform above tasks.
- In addition to the bean registration the managing the life cycle of the Beans and ***it allows you to perform custom tasks at a particular point of their life cycle.*** Your tasks should be encapsulated in callback method for the IOC container to call any suitable time.
- The spring IOC container manages a life cycle of a bean and also it will take the following responsibility-

1. To create a Bean interface either by a constructor or a factory method.
2. Set the values and Bean references to the Bean properties.
3. To call the initialization and callback methods.

4. To make a Bean ready to use
5. When the Container is shut down, it calls the destruction callback() method.

- Initializing Bean interface is present in "org.springframework.beans.factoryInitializingBean" interface.
- This interface contain only one callback method afterPropertySet() method.
  public void afterPropertySet() throws Exception
- DisposalBean interface is presen in "org.springframework.bean.factory" package.
- It has one callback method.
  public void destroy() throws Exception
- We can set the init() and destroy() method attributes in the Bean declaration and specify the callback method name.
- In spring 2.5 or later, we can also annotate initialization and destruction callback method within the life cycle annotation @Postconstruct and @Predestroy.
- To understand how the sprig IOC container manages the life cycle of a spring Beans let us consider the example –

SpringBeanLifeCycle

```
|------src
        |------default package
        |       |------Cashier.java
        |       |-------Product.java
        |       |-------ShoppingCart.java
        |       |-------Test.java
        |-------log4j
        |-------myBeans.xml
```

//Product.java

```
public class Product{
        int pid;
        String name;
        double price;
        //required setter and getter
```

```
}

//ShoppingCart.java
public class ShoppingCart{
        List<Product> items;
        //required setter and getter
}

//Cashier.java
public class Cashier implements initializingBean, DisposableBean{
        private static Logger logger = Logger.getLogger(Cashier.class);
        private String name;
        private String path;
        private PrintWriter writer;

        public void setName(String name){
                logger.info("injection happened in name property of Cashier by using
                                                                        setName()");
                this.name = name;
        }
        public void setPath(String path){
                logger.info("injection happened in path property of Cashier by using
                                                                setPath()");
                this.path = path;
        }
        public void openFile() throws Exception{
                logger.info("openFile() method executed and created a file in the
                                        following location:"+path+":"+name);
                File logFile = new File(path, name+".txt");
                writer = new PrintWriter(new OutputStreamWriter
                                                (new FileOutputStream(logFile,true)))
        }
        public void checkout(ShoppingCart car)throws IOException{
                logger.info("Business method checkout start");
                String name = "";
```

```
            for(product product:cart.getItem()){
                name+ = product.getName()+" ";
            }
            writer.flush();
            logger.info("Business method checkout ended");
        }
        public void closeFile()throws IOException{
            logger.info("close file method of cashier");
            writer.close();
        }
        @Override
        public void destroy()throws IOException{
            logger.info("callback destruction method executed");
            openFile();
        }
        public void afterPropertiesSet() throws IOException{
            logger.info("callback ethod initialization method executed");
            openFile();
        }
}
```

//myBeans.xml

```
<beans>
<bean id = "cashier1" class = "Cashier">
<property name = "name" value = "cashier"/>
<property name = "path" value = "E:logFile"/>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
            ClassPathXmlApplicationContext context =
                        new ClassPathXmlApplicationContext("myBeans.xml");
            Cashier cashier = (Cashier)context.getBean("cashier1");
```

```
            Product p1 = new Product();
            p1.setPid(1001);
            p1.setName("mouse");
            p1.setPrice(3000);
            Product p2 = new Product();
            p2.setPid(1002);
            p2.setName("Keyboard");
            p2.setPrice(4000);
            List list = new ArrayList();
            list.add(p1);
            list.add(p2);
            ShoppinCart cart = new ShoppingCart();
            cart.setItem(list);
            cashier.checkOut(cart);
            context.destroy();
        }
}
```

Note:

Whenever we are implementing InitializinBean and DisposableBean interface, our bean class not a POJO. So a better approach of specifying the initialization and destruction callback method and destroy() attribute in bean declaration.

```
<bean id = "cashier" class = "Cashier" init-method= "openFile"
destroy-method = "closeFile">
<property name = "name" value = "cashier1"/>
<property name = "path" value = "E:logFile"/>
</bean>
```

Note:

Both BeanFactory and ApplicationContext container support the life cycle methods, but we can't stop the BeanFactory container explicitly to destroy the singleton beans at the end of the client application. We can call factory.destroySingletons(), if the container is BeanFactory.

Ex-

```
//Test.java
class Test{
        public static void main(String[] args){
                XmlBeanFactory factory = new XmlBeanFactory
                                        (new ClassPathresource("myBeans.xml");
                ……….
                ……….
                Factory.destroySingletons();
        }
}
```

If you are working with ApplicationContext container, it is possible to shut down the container using context.destroy() method at the end of client application.

Ex-

```
class Test{
        public static void main(String[] args){
                ClassPathXmlApplicationContext context =
                                new ClassPathXmlApplicatinContext("myBeans.xml");
                ……….
                ……….
                Context.destroy();
        }
}
```

Interface Injection:

- If the spring container performs the dependency injection on Bean properties by implementing getXxx() method Aware interface methods, then it is called as interaface injection.
- Interface injection is also useful to make the spring bean to know about itself and after bean managed by the container.
- All the xxxAware interface's are predefinec interfaces supplied by spring API.

- We can perfor 3 types of dependency injection. We can't use interface to injection to inject our own values.
- Interface injection can perform special values injection given by spring container.

BeanNameAware Interface:

- This interface present in "org.springframework.beans.factory".
- This interface contain one method-
  pubic void setBeanName(String name)

BeanFactoryAware interface:

- This interface present in "org.springframework.factory" package.
- It is used to get the container information and container object reference.
- It has one method –
  public void setBeanFactory(BeanFactory factory)

ApplicationContextAware Interface:

- This interface present in "org.springframework.context" package.
- It has one method-
  public void setApplicationContext(ApplicationContext context)
- This interface is also used to get the container object reference and container interface (ApplicationContextContainer).

Note:

While implementing the interface injection our spring bean is not a POJO.

InterfaceInjection

|------default package
  |-----MyBean.java
  |------Test.java
  |------myBeans.xml

//MyBean.java

```
public class MyBean implements BeanNameAware{
        private ApplicatonContext context;
        private String BeanName;
        public void setApplicationContext(ApplicationContext context)
                                                throws BeansException{
                this.context = context;
        }
        public void setBeanName(String name){
                this.beanName = name;
        }
        public void getDetails(){
                System.out.println("The current BeanName is :"+beanName);
                System.out.println("is Singleton Bean :"+
                                                context.isSingleton(beanName));
                System.out.println("is prototypeBean :"+
                                                    context.isPrototype(beanName);
                System.out.println("Current no of bean are managed by container :"+
                                        context.getBeanDefinationCount());
                System.out.println("Other Bean managed by container : "+
                                        Context.getBean(date1);
        }
}
```

//myBean.xml

```
<beans>
<bean id  = "myBean" class = "MyBean"/>
<bean id = "date1" class = "java.util.Date"/>
<bean id = "date2" class = "java.util.Date"/>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                ApplicationContext context = new ClassXmlApplicationContext
                                                        ("myBeans.xml");
```

```
        MyBean myBean = (MyBean)context.getBean("myBean");
        myBean.getDetails();
    }
}
```

BeanPostProcessor:

- It is an interface present in ***org.springframework.beans.factory.config*** package.
- The interface contains the following two method –

1. public void postProcessBeforeInitialization(Object bean, String beanName);
2. public void postProcessAfterInitialization(Object bean, String beanName);

- BeanPostProcessor is used to provide some common logic to all the beans or set of beans which are configured in the spring configuration file.
- ***The BeanPostProcessor allows additional bean processing before and after initialization call back method.***
- The main characteristic of a BeanPostProcessor is that it will process all the bean instances in IOC container one by one not just a single bean instance.
- postProcessBeforeInitialization() method is executed just before the call back initialization method call.
- postProcessAfterInitialization() method is executed just after the call back initialization method call.

Note:

These methods are called for each spring bean which are configured in spring configuration file.

SpringBeanLifeCycleWithBeanPostProcessorObject

```
|------src
    |------(default package)
        |-------Cashier.java
        |-------PathCheckingBeanPostProcessor.java
```

|-------Product.java → Same as SpringBean life cycle
|-------ShoppingCart.java → Same as SpringBean life cycle
|-------StorageConfig.java
|-------Test.java → Same as SpringBean life cycle
|------log4j.properties

//PathCheckingBeanPostProcessor.java

```
public class PathCheckingBeanProcessor implements BeanPostProcessor{
        private static Logger logger =
                        Logger.getLogger(PathCheckingBeanPostProcessor.class);
        public Object postProcessAfterInitialization(Object bean, String beanName)
                                                                throws BeanException{
                logger.info("postProcessAfterInitialization:"+beanName);
                return bean;
        }
public Object postProcessBeforeInitialization(Object bean, String beanName)
                                                        throws BeanException{
                logger.info("postProcessBeforeInitialization:"+beanName);
                if(bean instanceof StorageConfig){
                        StorageConfig config = (StorageConfig)bean;
                        String path = config.getPath();
                        File file = new File(path);
                        if(!=file.exists()){
                                file.mkdir();
                        }
                }
                return bean;
        }
}
```

//StorageConfig.java

```
public interface StorageConfig{
        String getPath();
}
```

```
//Cashier.java
public class Cashier implements StorageConfig{
        private static Logger logger = Logger.getLogger(Cashier.class);
        private String name;
        private String path;
        private PrintWriter writer;

        public void setName(String name){
                logger.info("injection happened by name property of Cashier by using
                                                                        setName()");
                this.name = name;
        }
        public void setPath(String path){
                logger.info("injection happened by name property of Cashier by using
                                                                        setPath()");
                this.path = path;
        }
        @PostConstruct     //this annotation is used to make a method as a callback
                           //initialization method
        public vod openFile() throws IOException{
                logger.info("openFile() method executed and created a file in the
                                        following location:"+path+":"+name);
                File logFile = new File(path, name+".txt");
                Writer = new PrintWritter(new OutputStreamWritter(new
                                                FileOutputStream(logFile,true)));
        }
        public void checkout(ShoppingCart cart)throws IOException{
                logger.info("Businees method checkout started");
                String name = "";
                for(Product prodect: cart.getItems()){
                        name+product.getName()+" "+;
                }
        writer.println(new Date()+"\t"+name+"\r\n");
        writer.flush();
```

```
        logger.info("Busineess Method checkout ended");
        }
        @PreDestroy         //makes the method as a callback destroy method
        public void closFile() throws IOException{
                logger.info("closeFilel Method of cashier");
                writer.close();
        }
        public String getPath(){
                return path;
        }
}// end of cashier
```

//configuration File

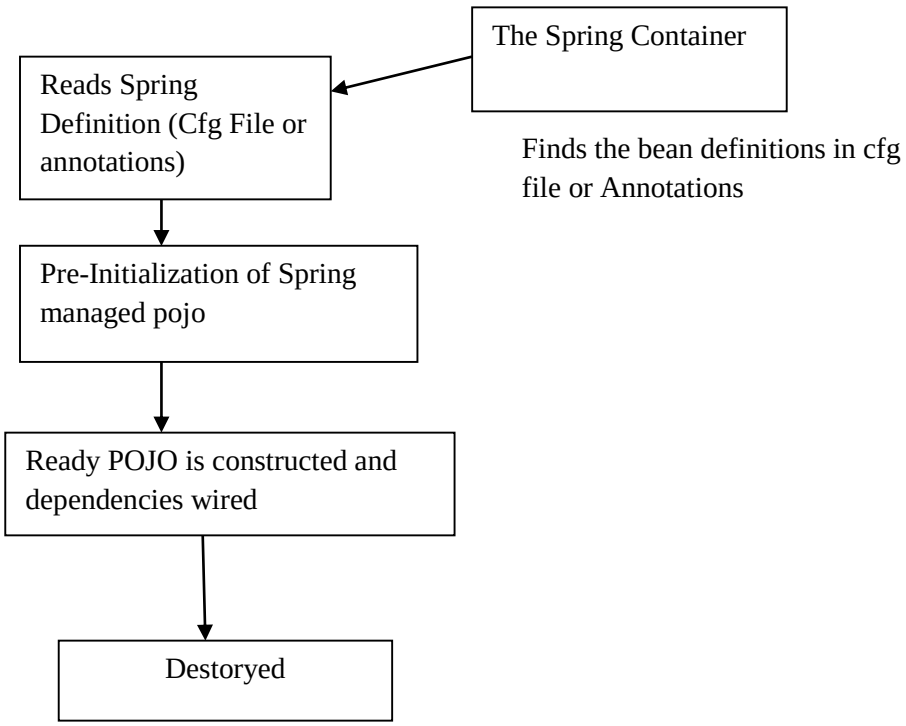
```
<beans XSD….>
<context: annotation-config/>
<bean class = "PathCheckingBeanPostProcessor"/>
<bean id = "cashier1" class = "Cashier">
<property name = "name" value = "cashier1"/>
<property name = "path" value = "G://CashierData"/>
</bean>
</beans>
```

- During the above Cashier Bean instantiation the following operations are performed by spring IOC containser-

    1. Cashier bean instance is created by using Reflection-API.
    2. After setName() and setPath() methods are executed to prove the initialization on Cashier bean properties (i.e. name, path).
    3. After PathCheckingBeanPostProcessor object postProcessBeforeInitialization() method will be executed. By this JVM is checking the given path is existed or not.
    4. If the path is not existed, then directory is created based on the given path.

5. Just after callback initialization method (i.e. openFile() method) will be executed. (File is created and also PrintWriter stream object is created).
6. PathCheckingBeanPostProcessor object postProcessAfterInitialization method will be executed.
7. Cashier bean instance is ready for use (business method checkout () is called by the container).
8. Closing the container object. At this time bean object callback destruction method (i.e. closeFile() method) is executed.

Spring Bean Life Cycle Diagram:



- Spring IOC container creates an instance of the spring bean by using java Reflection-API.
- If any properties are mentioned, if the setter injection is configured, then container will call setXXX() methods.
- If the bean class implements BeanNameAware interface, then container calls setBeanName() method.

- If the bean class implements the ApplicationContextAware interface, then container calls setApplicationContext() method.
- If the bean class implements the BeanFactoryAware interface, then container calls setBeanFactory() method.
- If any beanPostProcessors are associated with the container, then container will call postProcessBeforeInitialization() method.
- If the bean class implements the InitializingBean interface OR any custom init() method is declared OR any method is annotated with @PostConstruct annotation, then container calls that callback initialization methods.
- If any bean postPorcessors are associated with the container, then container calls postProcessAfterInitialization() method.
- The bean is initialized and ready to be used.
- If the bean class implements Disposable interface OR any custom destroy method is declared OR any method is annotated with @PreDestroy annotation, then container calls callback destruction methods.

Q. what is container callback and container life cycle?
Ans. A method that is called by underlying container automatically where event that is rise is called container callback method or container life cycle method.

Inner Bean:

- A spring bean is configured within the <***property***> tag or <***constructor-arg***> tag by using bean tag is known as Inner Bean configuration or Inner Bean.
- While configuring the Inner bean name or id attributes are not useful, so we can say Inner bean also called anonymous bean.

InnerBeanProject

```
|-------src
        |-------defaut package
        |       |--------FishBean.java
        |       |-------Test.java
        |       |-------WaterBean.java
        |-------myBeans.xml
```

//FishBean.java

```
public class FishBean{
        public FishBean(){
                System.out.println("FishBean constructor");
        }
        public void fishBeanMethod(){
                System.out.println("Fish Bean Method");
        }
}
```

//WaterBean.java

```
public class WaterBean{
        private FishBean fishBean;
        public void setFishBean(FishBean fishBean){
                this.fishBean = fishBean;
        }
        public WaterBean(){
                System.out.println("WaterBean Constructor");
        }
        public void waterBeanMethod(){
                System.out.println("WaterBean Method");
                fishBean.fishBeanMethod();
        }
}
```

//myBeans.xml

```
<beans>
<bean id = "wb" class = "WaterBean">
<property name = "fishBean">
<bean class = "FishBean"/>
</property>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args(){
                String configfile = "myBeans.xml";
                ApplicationContext context =
                                        new ClassPathXmlApplicationContext(configfile);
                WaterBean w = (WaterBean)context.getBean("wb");
                w.waterBeanMethod();
        }
}
```

Lazy init attribute:

- When are using ApplicationContext based container by default the container creates all the beans which are the configured in the spring configuration file at the time of loading the configuration file, but sometimes we don't want to create the beans until and unless the request comes (call of getBean()) for a particular bean.
- This can be achieved with the ***lazy-init*** attribute of <bean> tag.

Ex –

```
<beans>
<bean id = "emp" class = "com.nareshit.Employee"
lazy-init = "true">                    |-------the Employee object will create whenever
</bean>                                        we are calling getBean("emp") method
```

</beans>

Multiple No. of Configuration Files in Spring Application:

- When the spring configuration file size is huge, it is difficult to maintain, so in that scenarios we generally have multiple configuration files.
- We can work with the multiple configuration file in two ways –

1. By importing other spring configuration file into the current configuration file.
   Ex –
   <import resource = "other-spring-beans.xml"/>

2. By passing all the spring bean configuration files as a String array to the constructor of the container.
   Ex –
   ApplicationContext context = "new classPathXmlApplicationContext
   (new String[]{"beans1.xml","beans2.xml","beans3.xml"});

MultiConfigExample

```
|-------src
        |------com.nareshit.client
                |------Test.java
        |------com.nareshtit.controller
                |------Controller.java
        |------com.nareshit.dao
                |-------DAO.java
        |------com.nareshit.service
                |-------Service.java
        |------com.nareshit.xml
                |------controllerBeans.xml
                |------daoBeans.xml
                |------serviceBeans.xml
```

```
//Controller.java
public class Controller{
        private Service service;
// required setter and getter
        public void controllerMethod(){
                service.serviceMethod();
                System.out.println("Controller Method");
        }
}
//Service.java
public class Service{
        private DAO dao;
        //required setter and getter
        public void serviceMethod(){
                dao.daoMethod();
                System.out.println("Service Method");
        }
}

//DAO.java
public class DAO{
        public void daoMethod(){
                System.out.println("DAO Method");
        }
}

//controllerBeans.xml
<beans>
<import resource = "serviceBeans.xml"/>
<bean id = "controllerObj" class = "com.nareshit.controller.Controller">
<property name = "service" ref = "serviceObj"/>
</bean>
</beans>
```

//serviceBeans.xml

```
<beans>
<import resource = "daoBeans.xml"/>
<bean id = "serviceObj" class = "com.nareshit.service.Service">
<property name = "dao" ref = "daoObj"/>
</bean>
</beans>
```

//daoBeans.xml

```
<beans>
<bean id = "doaObj" class = "com.nareshit.doa.DAO">
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                String configfile = "com/nareshit/xml/controllerBeans.xml";
                ApplicationContext context =
                                new ClassPathXmlApplicationContext(configfile);
                Controller c = (Controller)context.getBean("controllerObj");
                c.controllerMethod();
        }
}
```

Difference b/w id and name attribute:

- Id attribute is used to hold the object name, name attribute is used to hold the alias names.
- By using name attribute is possible to provide more than one name,
- While getting the bean object from container, we can id attribute name or name attribute name.

Ex-

<bean id = “std”, name = “std1, std2, std3” class = “Student”>

In Client App

Student s = (Student)container.getBean(“std”);

OR

Student s1 = (Student)container.getBean(“std1”);

Q. How to read data from .properties file to sprig bean property?

Ans.

To make the ApplicationContext container recognizing a .properties file, we need to configure a special spring bean called ***org.springframework.beans.factory.config.PropertyPlaceHolderConfigurer*** by specifying name of properties file.

Note:

The text file that maintain the entries in the form of key-value pairs is called as text properties file.

Ex-

Myfile.properties

#properties file(comment)

name = 3sha

age = 30

mobile = 9878243567

“Here name and are keys and 3sha and 30 are values”.

In JDBC applications we use properties file to pass JDBC driver details and database details to get the flexibility on modifications.

Note:

Spring application can’t directly communicate with the properties file, it is done by using XML file.

Syntax of the placehoder:-

${<key name>}

Note:

BeanFactory containser can't work with .properties file.

ApplicationContext container can work with .properties file.

ExternalPropertiesFileExample

```
|-------src
        |------com.nareshit.client
                |----Test.java
        |------com.nareshit.jdbc
                |------DatabaseConnectionDetails.java
        |------db.properties
        |------db1.properties
        |------myBeans.xml
```

//DatabaseConectionDetails.java

```
public class DatabaseConectionDetails{
        private String userName;
        private String url, password, driverClass;
//required setter and getter
}
```

//db.properties

```
#Oracle databaseConnectionDetails
user = system
password  = manager
url = jdbc:oracle:thin:@localhot:1521:XE
driverClass = oracle.jdbc.driver.OracleDriver
```

//db1.proerties

```
user  = 3sha
password = 9tra
```

//myBeans.xml

```
<beans>
<bean class =
 “org.springframework.beans.factory.config.PropertyPlaceholderconfigurer”>
<property name = “Locations”>
<list>
<value>db.properties</value>
<value>db1.properties</value>
</list>
</property>
</bean>
<bean id = “dbConnection” class = “con.nareshit.DatabaseConnectionDetails”>
<property name = “username” value = “${user}”/>
<property name = “password” value = “${password}”/>
<property name = “url” value = “${url}”/>
<propert name = “driverClass” value = “${driverClass}”/>
</bean>
</beans>
```

//Test.java

```
public class Test{
        public static void main(String[] args){
                ApplicationContext context = new ClassPathXmlApplicatonContext
                (“myBeans.xml”);
                DatabaseConnectoinDetails dbConnection =
                        (DatabaseConnectionDetais)context.getBean(“dbConnection”);
                System.out.println(“UserName:”+dbConnection.getUserName());
                System.out.println(“Password:”+dbConnection.getPassword());
                System.out.println(“DriverClass:”+dbConnection.getDriverClass());
                System.out.println(“URL:”+dbConnection.getUrl());
        }
```

}

Spring Annotations:

- Normally we can declare all the bean or components in XML bean configuration file, so that spring container can detect and register your beans or components.
- But actually, Spring container able to auto scan, detect and instantiate your beans from project package.

Auto Component Scanning:

- To enable the spring auto component scanning feature annotate the classes with ***@Component*** annotation.
- This annotation indicate to the container this class is an auto scan component.

Ex-

```
import org.springframework.beans.stereotype.Components;
@Component
public class EmployeeService{
}
```

- Custom auto scan component name by default, Spring Container will create with "EmployeeService" to "employeeService" i.e. lower case of the first character of the component (with the same Component Name).
- We can retrieve this component name with "employeeService" -

```
EmployeeService emp = (EmployeeService)context.getBean("employeeService");
```

- To create a custom name for component we can work with @Service, @Repository, @Controller annotations.

  Ex-

  ```
  @Servie("AAA")
  public class EmployeeService{
  }
  ```

Now you can retrieve
EmployeeService emp = (EmployeeService)context.getBean("AAA");

Note:

In Spring 2.5 there are 4 types of auto components scan annotation types.

@Component – indicates an auto scan component.
@Repository – indicates DAO component in the persistence layer.
@Service – indicates a Service component in the business/service layer.
@Controller – indicates a controller component in the presentation layer.

Internal code of @Repository annotation:

```
 @Target{ElementType, TYPE}
@Retention(RetentionPolicy, RUNTIME)
@Documented
@Component
public @interface Repository{
        String value () default "";
}
```

Internal code of @Service annotation:

```
 @Target{ElementType, TYPE}
@Retention(RetentionPolicy, RUNTIME)
@Documented
@Component
public @interface Service{
        String value () default "";
}
```

Note:

@Repository, @Service and @Controller annotations are annotated with @Component. To increase the readability of the program we should always declare @Repository, @Service and @@controller for a specified layer.

Note:

To auto-scan to components in the configuration file we can use the following tag –

<context:component-scan base-package = “com.nareshit”/>

|-----default value is default package

SpringAnnotationExmple

```
|------src
        |-------com.nareshit.client
        |       |--------Test.java
        |-------com.nareshit.dao
        |       |-------EmployeeDAO.java
        |-------com.nareshit.service
        |       |--------EmployeeService.java
        |-------myBeans.xml
```

//EmployeeService.java

```
package com.nareshit.service
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import org.nareshit.dao.EmployeeDAO;
@Service(“eservice”)
public class EmployeeService{
        @Autowired
        @Qualifier(“edao”)
        Private EmployeeDAO employeeDao;
        public String toString(){
                return “this is Employee Service: “+employeeDao;
        }
}
```

//EmployeeDAO.java

```
package com.nareshit.dao;
import org.springframework.stereotype.Repository;
@Repository("edao")
public class EmployeeDAO{
        public String toString(){
                return "EmployeeDAO";
        }
}
```

//myBeans.xml

```
<beans>
<context:component-scan base-package = "com.nareshit"/>
```

//Test.java

```
public class Test{
        public static void main(Stringp[] args){
                ApplicationContext context =
                                new ClassPathXmlApplicationContext("myBeans.xml");
                EmployeeSerivce eservice =
                                        (EmployeeeService)context.getBean("eservice");
                System.out.println(eservice);
        }
}
```