**Framework:**

* It is a special software that is capable of developing applications based on ceratian architecture with the ability to generate common logics of the application.
* It is given based on core technologies.
* It provides an abstraction layer on core technologies.

**Struts:**It is a framework to develop MVC-II architecture based web applications only.

**Hibernate:**Its a ORM Framework which is used only for database operations.

**Spring:**

Its a framework software which can be used to develop any kind of Java/J2EE applications.  
Spring framework provides abstraction layer on:

* java & J2EE core technologies
* ORM Tools like:
  1. Hibernate
  2. JDO
  3. iBastis etc..,
  4. AOP Framework etc..,

**Introduction:**

1. Spring framework is developed to simplify the development of enterprise applications in Java technologies.
2. It is an open source framework begin developed by Interface21.
3. The Spring provides light weight IoC Container and AOP framework.
4. It provides support for JPA, Hibernate, Web services, Schedulers, Ajax, Struts, JSF and many other frameworks.
5. The Spring MVC components can be used to develop MVC based web applications.
6. The Spring 3.0 framework has been released with major enhancements and support for the Java 5 [JDK1.5].
7. Spring can be used to configure declarative transaction management, remote access to your logic using RMI or web services, mailing facilities and various options in persisting your data to a database.
8. Spring framework can be used in modular fashion, it allows to use in parts and leave the other components which is not required by the application.

* Spring is an open source framework, which is very flexible while developing any kind of application. i.e. stand alone application/web-application/enterprise applications.
* Struts is used only for web applications, where as Spring can be used for developing stand-alone application, applet based application, web application and enterprise application.
* Spring framework is created to address the complexity of enterprise application development.
* One of the chief advantages of the Spring framework is its layered architecture
* Spring is a light-weight framework.
* Spring is having total 6 modules.
* We can use the modules independently or combine depending on the type of application and requirement.
* Main core of Spring is IOC module.

**The Advantages of spring framework**

The advantages of spring are as follows:

* Spring has layered architecture. Use what you need and leave you don't need now.
* Spring Enables POJO Programming. There is no behind the scene magic here. POJO programming enables continuous integration and testability.
* Dependency Injection and Inversion of Control Simplifies JDBC
* Open source and no vendor lock-in.

**The Features of Spring framework**

The features of spring framework are as follows

* **Lightweight:**Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.
* **Inversion of control (IOC):**Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.
* **Aspect oriented (AOP):**Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.
* **Container:**Spring contains and manages the life cycle and configuration of application objects.
* **MVC Framework:**Spring comes with MVC web application framework, built on core Spring functionality. This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.
* **Transaction Management:**Spring framework provides a generic abstraction layer for transaction management. This allowing the developer to add the pluggable transaction managers, and making it easy to demarcate transactions without dealing with low-level issues. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.
* **JDBC Exception Handling:**The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy. Integration with Hibernate, JDO, and iBATIS: Spring provides best Integration services with Hibernate, JDO and iBATIS

Spring framework came up with every module for every tier. No other framework contains these many modules.  
  
Introduction Read more: [http://java.spring.jobs4times.com/spring-.html#Introduction#ixzz4OVQO6E2X](http://java.spring.jobs4times.com/spring-.html#ixzz4OVQO6E2X)

**Spring Architecture:**

Spring is well-organized architecture consisting of seven modules.   
Modules in the Spring framework are:

Spring1.x: It has 7 modules. They are:

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| AOP Module | |  |  | | --- | --- | | O/R Mapping Module | Web Context and Utility Module | | JDBC and DAO Module | Application Contex Module | | MVC Module |

|  |
| --- |
| Core Container and Supporting Utilities |

**Spring2.x:**It has 6 modules only. Because Web & Web MVC modules were clubbed together & given as Web MVC module.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | O/R Mapping Module | JDBC and DAO Module |  |  | | --- | | AOP Module | | JEE/Contex Module | Web  &  MVC Module |

|  |
| --- |
| Core Container and Supporting Utilities |

**Core Module:**

1. It is base module for all modules.
2. Provides BeanFactory container which makes Spring as a container
3. provides the fundamental functionality of the Spring framework.
4. The Core package is the most import component of the Spring Framework.
5. This component provides the Dependency Injection [DI] features.

This module contains IOC container.

IOC container is a piece of s/w, Which creates the objects and establish the dependencies.

package com.pioneercoders.beans;

public class StudentBean {

//steps to design spring core helloworld program

/\*

\* 1.Add the dependencies for Spring framework

\* 2.Create the bean class(Student bean with variable and setter method to set the data ) ,because spring treat our class as bean

3.Create the application-context.xml file and mention our where our bean class are configured including package location (com.pioneercoders.beans.StudentBean)

\* 4.create the Ioc container(using beanfactory or applicationcontext ) and get object from application-context-file

\*

\*/

private String message;

public void setMessage(String message) {

this.message = message;

}

public void displayMessage(){

System.out.println("Your message is"+" " +message);

}

}

package com.pioneercoders.test;

public class SpringTheory {

/\*

\* Spring is versatile framework which we can develop easily javaee applications ,and It was developed by Rod Johnson in 2003.

\*

\* Spring will have several modules like spring-core ,spring-jdbc,spring-aop,spring-mvc with this we can develop a fulfledged web application without using help of other framework or if we want also spring people will give flexibility to work with any technology

\*

\* The other moto spring framework is to avoid tightly coupling between the classes

\*

\* If we want to use the features of other class compulsory we have to inherit ,but in this we will face so many problems if parent class changes child class also get effected

\*

\* along with this some many problems are there ,to avoid tightly coupling between the classes they will introduct IOC (Inverse of control )is a design pattern that will do following activities

\*

\*

\* The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

to instantiate the application class

to configure the object

to assemble the dependencies between the objects

\*

\*

\*

\*

\*

\*

\* Generally spring will treat our class as bean we will configure our class in application-context.xml(not standard name) and IOC container reads the xml configuration file (our class information) creates the object of our bean class and manages it until bean destruction

\*

\*

\* How many ways we can create IOC container

\*

\* Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

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

BeanFactory factory=new XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

\*

\*

\*

\*/

}

package com.pioneercoders.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.beans.StudentBean;

public class StudentApplicationContextTest {

public static void main(String[] args) {

//IOC container reads the configuration information from application-context.xml

//call the IOC container by using Bean Factory or Application Context

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

//after reading configuration file IOC creates the object for our class and

//i'm not creating object IOC container already creates the object we are getting that reference

//application-context will several beans which bean we are getting typecast it

StudentBean sb=(StudentBean)context.getBean("studentBean");

//we can set the data for our variables in configuration also by using dependency injection it can inject values

//we can inject values either by using constructor(constructor injection) or using setter methods (setter injection)

sb.setMessage("gud mng welcome to spring core ");

sb.displayMessage();

}

}

package com.pioneercoders.test;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import com.pioneercoders.beans.StudentBean;

public class StudentBeanFactoryTest {

public static void main(String[] args) {

//calling the IOC container using BeanFactory

Resource resource = new ClassPathResource("application-context.xml");

BeanFactory factory = new XmlBeanFactory(resource);

StudentBean sb =(StudentBean)factory.getBean("studentBean");

sb.setMessage("Ioc is created using Beanfacory");

sb.displayMessage();

}

}

Application-context.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="studentBean" class="com.pioneercoders.beans.StudentBean">

</bean>

</beans>

package com.pioneercoders.beans;

public class Course {

private int courseid;

private String coursename;

private String courseduration;

public void setCourseid(int courseid) {

this.courseid = courseid;

}

public void setCoursename(String coursename) {

this.coursename = coursename;

}

public void setCourseduration(String courseduration) {

this.courseduration = courseduration;

}

public void registerCourse(){

System.out.println("Registered with "+" "+courseid+" "+coursename+" "+courseduration);

}

}

package com.pioneercoders.dependentobjecttest;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import com.pioneercoders.setterdependentobject.Employee;

public class SetterDependencyObjectTest {

public static void main(String[] args) {

//call IOC container

Resource r=new ClassPathResource("context.xml");

BeanFactory factory=new XmlBeanFactory(r);

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

e.displayInfo();

}

}

package com.pioneercoders.setterdependentobject;

public class Address {

private String addressLine1,city,state,country;

//getters and setters

public String toString(){

return addressLine1+" "+city+" "+state+" "+country;

}

public void setAddressLine1(String addressLine1) {

this.addressLine1 = addressLine1;

}

public void setCity(String city) {

this.city = city;

}

public void setState(String state) {

this.state = state;

}

public void setCountry(String country) {

this.country = country;

}

}

package com.pioneercoders.setterdependentobject;

//we can inject not only primitives but also we can inject one bean into another bean

//Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object.

public class Employee {

private int id;

private String name;

private Address address;

//setters and getters

public void displayInfo(){

System.out.println(id+" "+name);

System.out.println(address);

}

public void setId(int id) {

this.id = id;

}

public void setName(String name) {

this.name = name;

}

public void setAddress(Address address) {

this.address = address;

}

}

package com.pioneercoders.test;

public class DependencyInjectionEx {

/\* Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.

\* Dependency Injection makes our programming code loosely coupled.

\*

\* If we give our bean class details it will automatically inject values in to our beans

\*

\* We can achieve dependency using

\*

\* By Constructor( by passing parameters via constructor )

\*

\* we can inject values using <constructor-arg> tag in application-context.xml

\*

By Setter method (by generating setter method )

we can inject values using <property> tag in application-context.xml

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*/

}

package com.pioneercoders.test;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import com.pioneercoders.beans.Course;

public class SetterInjectionEx {

public static void main(String[] args) {

Resource resource = new ClassPathResource("application-context.xml");

BeanFactory factory = new XmlBeanFactory(resource);

Course c= (Course)factory.getBean("coursebean");

//by using setter injection we are injection values to Course class variables call registerCourse() method

c.registerCourse();

}

}

Context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="address1" class="com.pioneercoders.setterdependentobject.Address">

<property name="addressLine1" value="5-62,ongole"></property>

<property name="city" value="ongole"></property>

<property name="state" value="AP"></property>

<property name="country" value="India"></property>

</bean>

<bean id="obj" class="com.pioneercoders.setterdependentobject.Employee">

<property name="id" value="1"></property>

<property name="name" value="bhargav pillai"></property>

<!-- injecting address class into employee ref attribute of property elements is used to define the reference of another bean. -->

<property name="address" ref="address1"></property>

</bean>

</beans>

Application-context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="coursebean" class="com.pioneercoders.beans.Course">

<!-- The property element defines the setter injection. The value subelement of property will assign the specified value to the variables-->

<property name="courseid">

<value>20</value>

</property>

<property name="coursename">

<value>SpringFramework</value>

</property>

<property name="courseduration">

<value>six</value>

</property>

</bean>

</beans>

package com.pioneercoders.collectioninjection.beans;

//we can inject dependencies either by using Setter or Constructor injection

public class ShoppingCart {

private int cartid;

private String cartname;

public ShoppingCart(int cartid, String cartname) {

super();

this.cartid = cartid;

this.cartname = cartname;

}

public void display(){

System.out.println("Cart details are "+" " +cartid+" "+cartname);

}

}

package com.pioneercoders.collectioninjectiontest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.collectioninjection.beans.ShoppingCart;

public class CollectionInjectionTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

ShoppingCart shopcart = (ShoppingCart)context.getBean("cart");

shopcart.display();

}

}

package com.pioneercoders.dependencyobject.beans;

//injecting Address class variables in Employee class using Constructor Injection

public class Address {

private String city;

private String state;

private String country;

public Address(String city, String state, String country) {

super();

this.city = city;

this.state = state;

this.country = country;

}

public String toString(){

return city+" "+state+" "+country;

}

}

package com.pioneercoders.dependencyobject.beans;

/\*

\* If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address.

\* The Address class object will be termed as the dependent object

\*

\*

\*

\*

\*

\* Address is dependent object

\*/

public class Employee{

private int id;

private String name;

private Address address;//Aggregation

public Employee(int id, String name, Address address) {

super();

this.id = id;

this.name = name;

this.address = address;

}

public void show(){

System.out.println(id+" "+name);

System.out.println(address);

}

}

package com.pioneercoders.dependencyobject.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.dependencyobject.beans.Employee;

public class DependencyObjectTest {

public static void main(String[] args) {

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

Employee emp =(Employee)context.getBean("employee");

emp.show();

}

}

Context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="address" class="com.pioneercoders.dependencyobject.beans.Address">

<constructor-arg value="hyd"></constructor-arg>

<constructor-arg value="hyderabad"></constructor-arg>

<constructor-arg value="India"></constructor-arg>

</bean>

<bean id="employee" class="com.pioneercoders.dependencyobject.beans.Employee">

<constructor-arg value="12" type="int"></constructor-arg>

<constructor-arg value="bhargav"></constructor-arg>

<!-- Address bean class is injecting into Employee by using ref tag -->

<constructor-arg>

<ref bean="address"/>

</constructor-arg>

</bean>

</beans>

Application-context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="cart" class="com.pioneercoders.collectioninjection.beans.ShoppingCart">

<!-- The constructor-arg element invokes the constructor

The value attribute of constructor-arg element will assign the specified value.

The type attribute specifies that int parameter and String parameter constructor will be invoked.

-->

<constructor-arg value="10" type="int"></constructor-arg>

<constructor-arg value="amazon" type="String"></constructor-arg>

</bean>

</beans>

package com.pioneercoders.collectioninjectiontest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.collectioninjectionusingconstructor.Customer;

public class CollectionDependencyObjectTest {

public static void main(String[] args) {

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

Customer cust= (Customer)context.getBean("customer");

cust.displayProductInfo();

}

}

package com.pioneercoders.collectioninjectiontest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.collectioninjectionusingsetter.Student;

public class CollectionDependencyTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

Student st = (Student)context.getBean("student");

st.displayStudentCourseInfo();

}

}

package com.pioneercoders.collectioninjectionusingconstructor;

import java.util.List;

public class Customer {

private int customerid;

private String customername;

private List<Product> products;

public Customer(int customerid, String customername, List<Product> products) {

super();

this.customerid = customerid;

this.customername = customername;

this.products = products;

}

public void displayProductInfo(){

System.out.println("Customer details are "+" "+customerid+" "+customername);

for(Product product: products){

System.out.println("Products are "+product);

}

}

}

package com.pioneercoders.collectioninjectionusingconstructor;

public class Product {

private int productid;

private String productname;

private float productcost;

public Product(int productid, String productname, float productcost) {

super();

this.productid = productid;

this.productname = productname;

this.productcost = productcost;

}

@Override

public String toString() {

return "Product [productid=" + productid + ", productname="

+ productname + ", productcost=" + productcost + "]";

}

}

package com.pioneercoders.collectioninjectionusingsetter;

public class Course {

private int courseid;

private String coursename;

private String facultyname;

public int getCourseid() {

return courseid;

}

public void setCourseid(int courseid) {

this.courseid = courseid;

}

public String getCoursename() {

return coursename;

}

public void setCoursename(String coursename) {

this.coursename = coursename;

}

public String getFacultyname() {

return facultyname;

}

public void setFacultyname(String facultyname) {

this.facultyname = facultyname;

}

@Override

public String toString() {

return "Course [courseid=" + courseid + ", coursename=" + coursename

+ ", facultyname=" + facultyname + "]";

}

}

package com.pioneercoders.collectioninjectionusingsetter;

import java.util.List;

public class Student {

private int studentid;

private String studentname;

//we can inject dependency object as collections also

//in one student will join multiple courses

List<Course> courses;

public void displayStudentCourseInfo(){

System.out.println("Student details are"+" "+studentid+" "+studentname);

for(Course course :courses){

System.out.println(course);

}

}

public int getStudentid() {

return studentid;

}

public void setStudentid(int studentid) {

this.studentid = studentid;

}

public String getStudentname() {

return studentname;

}

public void setStudentname(String studentname) {

this.studentname = studentname;

}

public List<Course> getCourses() {

return courses;

}

public void setCourses(List<Course> courses) {

this.courses = courses;

}

}

Context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="product1" class="com.pioneercoders.collectioninjectionusingconstructor.Product">

<constructor-arg value="1"></constructor-arg>

<constructor-arg value="shoes"></constructor-arg>

<constructor-arg value="2000.0f"></constructor-arg>

</bean>

<bean id="product2" class="com.pioneercoders.collectioninjectionusingconstructor.Product">

<constructor-arg value="2"></constructor-arg>

<constructor-arg value="mobile"></constructor-arg>

<constructor-arg value="9000.0f"></constructor-arg>

</bean>

<bean id="customer" class="com.pioneercoders.collectioninjectionusingconstructor.Customer">

<constructor-arg value="111"></constructor-arg>

<constructor-arg value="Srinadh"></constructor-arg>

<constructor-arg>

<list>

<ref bean="product1"/>

<ref bean="product2"/>

</list>

</constructor-arg>

</bean>

</beans>

Application-context.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="course1" class="com.pioneercoders.collectioninjection.Course">

<property name="courseid" value="1"></property>

<property name="coursename" value="Scjp"></property>

<property name="facultyname" value="Durga"></property>

</bean>

<bean id="course2" class="com.pioneercoders.collectioninjection.Course">

<property name="courseid" value="2"></property>

<property name="coursename" value="CoreJava"></property>

<property name="facultyname" value="Nagoorbabu"></property>

</bean>

<bean id="student" class="com.pioneercoders.collectioninjection.Student">

<property name="studentid" value="1001"></property>

<property name="studentname" value="manoj"></property>

<!-- in this scenario dependent object will list of values(student can join multiple courses) we can inject list tag -->

<property name="courses">

<list>

<ref bean="course1"/>

<ref bean="course2"/>

</list>

</property>

</bean>

</beans>

package com.pioneercoders.autowiring;

public class AutoWiringTheory {

/\* Spring framework is built on dependency injection and we inject the class dependencies through spring bean configuration file.

Usually we provide bean configuration details in the spring bean configuration file and we also specify the beans that will be injected in other beans using ref attribute.

But Spring framework provides autowiring features too where we don’t need to provide bean injection details explicitly.

There are different ways through which we can autowire a spring bean.

1.autowire byName – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.

2.autowire byType – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.

3.autowire by constructor – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.

4.autowire by autodetect – If you are on Spring 3.0 or older versions, this is one of the autowire options available. This option was used for autowire by constructor or byType, as determined by Spring container. Since we already have so many options, this option is deprecated

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*/

}

package com.pioneercoders.autowiring;

public class Address {

private String city;

private String state;

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getState() {

return state;

}

public void setState(String state) {

this.state = state;

}

}

package com.pioneercoders.autowiring;

//by using autowiring injection we are injecting Address class in Student class

public class Student {

private String studentname;

private Address address;

public String getStudentname() {

return studentname;

}

public void setStudentname(String studentname) {

this.studentname = studentname;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

}

package com.pioneercoders.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.autowiring.Student;

public class AutoWireByTypeTest {

public static void main(String[] args) {

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

Student stu = (Student)context.getBean("student");

System.out.println(stu.getStudentname());

System.out.println(stu.getAddress().getState());

System.out.println(stu.getAddress().getCity());

/\* ByType scenario

\*

\* Student context.xml (only one Address class bean it will injects if more than one Address bean it will throws error it gets confused which Address class values have to inject

\*

\* Address address class="com.pioneercoders.autowiring.Address">

\*

\* Address class matches with Address in context.xml

\*

\*

\*

\*/

}

}

package com.pioneercoders.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.autowiring.Student;

public class AutoWireByNameTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

Student stu = (Student)context.getBean("student");

System.out.println(stu.getStudentname());

System.out.println(stu.getAddress().getState());

System.out.println(stu.getAddress().getCity());

//we are getting only address ref injected values but not address2 ref values byName means

// Student Class Address ref match with configuration file ref

/\*

\* Student class

\*

\* Address address --> id="address" in context file

\*

\*

\*/

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="address" class="com.pioneercoders.autowiring.Address">

<property name="city" value="hyderabad" />

<property name="state" value="Telangana" />

</bean>

<bean name="student" class="com.pioneercoders.autowiring.Student"

autowire="byType">

<property name="studentname" value="Manohar" />

</bean>

<!-- Spring Bean Autowire "byType" In case of byType autowiring, spring

container looks for the class type. If in our XML configuration, there is

more than one eligible candidate by class type for autowiring, the container

will through error. There are three scenarios that may happen in case of

by type autowiring. 1. If in the container, there is only one bean of required

class type then autowiring is performed. 2. If there is more than one bean

of same class type in the container, a fatal error is thrown and autowiring

is not performed. 3. If there is no bean of required class type in the container,

obviously no autowiring performed and also no error is thrown.

In this we are showing scenario one only one match class u will try with scenario two and three

In this i'm injecting Address bean in Student bean by using autowire mode byType if class type matches it will injects

-->

</beans>

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="address" class="com.pioneercoders.autowiring.Address">

<property name="city" value="hyderabad"/>

<property name="state" value="Telangana"/>

</bean>

<bean name="address2" class="com.pioneercoders.autowiring.Address">

<property name="city" value="banglore"/>

<property name="state" value="Karnataka"/>

</bean>

<bean name="student" class="com.pioneercoders.autowiring.Student" autowire="byName">

<property name="studentname" value="Manohar"/>

</bean>

<!-- i'm injecting address bean in student bean byName matches with Student bean Address bean ref (address)-->

<!-- In our XML configuration file, we have created two beans with name address and address2 which is using same Address class but has been initialized with different property values.

In our Student class, the property name for Address class is address. The bean student has been defined as autowire="byName".

So the autowiring in Student class, will be performed using address bean. -->

</beans>

package com.pioneercoders.innerbeans;

public class Order {

private String item;

private String price;

private String address;

public String getItem() {

return item;

}

public void setItem(String item) {

this.item = item;

}

public String getPrice() {

return price;

}

public void setPrice(String price) {

this.price = price;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

@Override

public String toString() {

return "Order [item=" + item + ", price=" + price + ", address="

+ address + "]";

}

}

package com.pioneercoders.innerbeans;

/\* In Spring framework, whenever a bean is used for only one particular property, it’s advise to declare it as an inner bean.

\* And the inner bean is supported both in setter injection ‘property‘ and constructor injection ‘constructor-arg‘.

\*

\* Inner beans are the beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean.

\*

\* In this example PaymentGateway, which requires Order object to be injected.

\*/

public class PaymentGateway {

private Order order ;

public PaymentGateway(Order order) {

super();

this.order = order;

}

public void getOrder(){

//System.out.println("Order details are "+" "+order.getItem()+" "+order.getPrice()+" "+order.getAddress());

System.out.println(order);

}

}

package com.pioneercoders.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.innerbeans.PaymentGateway;

public class InnerBeanTest {

public static void main(String[] args) {

String confFile = "application-context.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(confFile);

PaymentGateway gateway = (PaymentGateway) context.getBean("paymentGwBean");

gateway.getOrder();

}

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="paymentGwBean" class="com.pioneercoders.innerbeans.PaymentGateway">

<constructor-arg>

<!-- to represent inner bean no need of id here Order bean is inner bean

directly injecting in PaymentGateWay bean -->

<bean class="com.pioneercoders.innerbeans.Order">

<property name="item" value="HeadFirstBook" />

<property name="price" value="RS 330" />

<property name="address" value="Bangalore" />

</bean>

</constructor-arg>

</bean>

</beans>

package com.pioneercoders.beaninheritance;

/\* A spring bean definition contains lot of information like property values, constructor arguments, and container specific information like init and destroy method settings and so on.

\* Spring allows to inherit all these bean properties. A child bean definition can inherit configurations from its parent definition.

\* The child bean definition can override some values, or add new values.

\*

\* This example shows how to inherit parent bean values with in child bean. We have Student class, the parent bean sets the common value for the property called studentcollege .

\* Any other bean definitions will inherits it. And add additional values

\*

\*

\*/

public class Student {

private int studentId;

private String studentname;

private String studentcollege;

public int getStudentId() {

return studentId;

}

public void setStudentId(int studentId) {

this.studentId = studentId;

}

public String getStudentname() {

return studentname;

}

public void setStudentname(String studentname) {

this.studentname = studentname;

}

public String getStudentcollege() {

return studentcollege;

}

public void setStudentcollege(String studentcollege) {

this.studentcollege = studentcollege;

}

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentname="

+ studentname + ", studentcollege=" + studentcollege + "]";

}

}

package com.pioneercoders.inheritancetest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.beaninheritance.Student;

public class BeanInheritanceTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

Student s = (Student)context.getBean("childStudentBean");

Student stu= (Student)context.getBean("baseStudentBean");

//in this we will get all properties including baseStudentBean details studentcollege

System.out.println(s);

//Student [studentId=1016, studentname=Gowtham, studentcollege=jntuk]

// ------------------------------------------------

//in this we are getting only baseStudentBean details studentcollege only but no studentid and studentname details

System.out.println(stu);

// Student [studentId=0, studentname=null, studentcollege=jntuk]

}

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- in this reference baseStudentBean acting as Parent bean -->

<bean id="baseStudentBean" class="com.pioneercoders.beaninheritance.Student">

<property name="studentcollege" value="jntuk" />

</bean>

<!-- in this childStudentBean acting as child bean inheriting property called studentcollege(same for every student) and adding new properties like studentId and studentname (different from student to student)-->

<bean id="childStudentBean" parent="baseStudentBean">

<property name="studentId" value="1016" />

<property name="studentname" value="Gowtham" />

</bean>

</beans>

package com.pioneercoders.beanscopes;

public class BeanScopesTheory {

/\*

\* When defining a <bean> in Spring, you have the option of declaring a

\* scope for that bean. For example, To force Spring to produce a new bean

\* instance each time one is needed, you should declare the bean's scope

\* attribute to be prototype. Similar way if you want Spring to return the

\* same bean instance each time one is needed, you should declare the bean's

\* scope attribute to be singleton.

\*

\* Simply we can control the creation or instance of bean in

\* application-context by using bean scopes .In spring by default scope is singleton

\*

\* The Spring Framework supports following five scopes, three of which are

\* available only if you use a web-aware ApplicationContext.

\*

\* The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

Scope Description

singleton This scopes the bean definition to a single instance per Spring IoC container (default).

prototype This scopes a single bean definition to have any number of object instances.

request This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

session This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

global-session This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

\*/

}

package com.pioneercoders.beanscopes;

public class DisplayMessage {

private String message;

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

package com.pioneercoders.beanscopestest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.beanscopes.DisplayMessage;

public class SingleTonScopeTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("application-context.xml");

//calling first time

DisplayMessage messageone = (DisplayMessage)context.getBean("displaymessage");

messageone.setMessage("hello h r u ");

System.out.println(messageone.getMessage());

//calling second time

DisplayMessage messagetwo = (DisplayMessage)context.getBean("displaymessage");

//here i'm setting message but i will get message already bean is created we are using same instance displaymessage

//it we send multiple requests alos it will only once if the scope is singleton and retruns same instance for all requests

System.out.println(messagetwo.getMessage());

}

}

package com.pioneercoders.beanscopestest;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pioneercoders.beanscopes.DisplayMessage;

public class PrototypeScopeTest {

public static void main(String[] args) {

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

//calling first time

DisplayMessage firstmessage = (DisplayMessage)context.getBean("displaymessage");

//set the message

firstmessage.setMessage("hello h r u ");

System.out.println(firstmessage.getMessage());

//calling second time

//scope prototype means for second request it will create one more instance for second request

DisplayMessage secondmessage = (DisplayMessage)context.getBean("displaymessage");

/\*

\*

\* for second request it will create new instance without setting message we are calling getMessage() method we will get data as null

\* without setting message to secondmessage ref we will get null

\*

\* output:

\*

\* hello h r u

null

\*

\*/

System.out.println(secondmessage.getMessage());

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made.

if scope is prototype for each request it will create a new bean

To define a prototype scope, you can set the scope property to prototype in the bean configuration file -->

<bean id="displaymessage" class="com.pioneercoders.beanscopes.DisplayMessage"

scope="prototype">

</bean>

</beans>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition.

it stores in cache area and returns same instance for multiple requests also

The default scope is always singleton however, when you need one and only one instance of a bean, we set the scope property to singleton in the configuration file -->

<bean id="displaymessage" class="com.pioneercoders.beanscopes.DisplayMessage"

scope="singleton">

</bean>

<!-- if we call any number times also it will creates single instance and returns the same instance -->

</beans>