**Spring Framework**

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

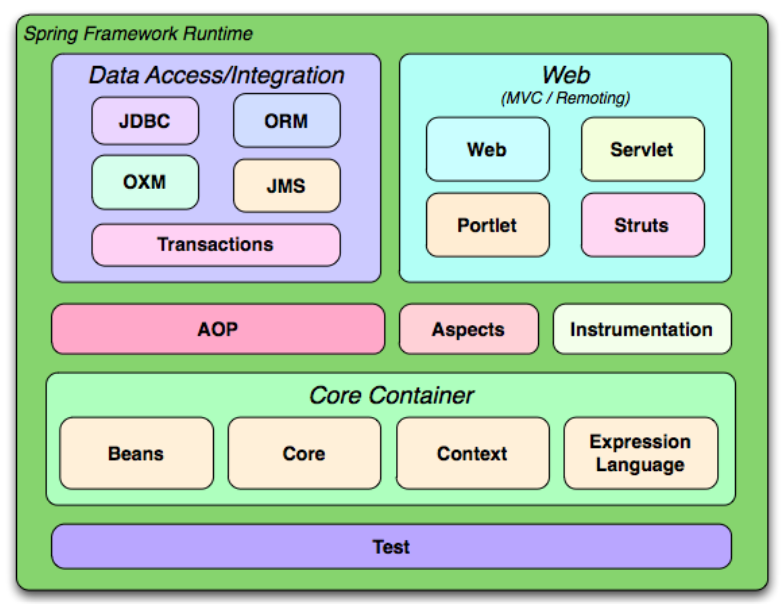
Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

JMS API: "The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients."

JMX API: "Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks."

* You would use JMS when you are building an system that needs (reliable, robust, resilient) message passing between different components (typically) on different computers.
* You would JMX when you are implementing monitoring for your system.

Spring Framework Overview:



# Spring – Understanding Inversion of Control.

Spring IoC (Inversion of Control) Container is the core of [Spring Framework](https://www.geeksforgeeks.org/introduction-to-spring-framework/). It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.  
**There are 2 types of IoC containers:**

* [BeanFactory](https://www.geeksforgeeks.org/spring-beanfactory/)
* [ApplicationContext](https://www.geeksforgeeks.org/spring-applicationcontext/)

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

# I. Spring – BeanFactory:

**Beans** are Java objects that are configured at run-time by [Spring IoC Container](https://www.geeksforgeeks.org/spring-ioc-container/). BeanFactory represents a basic**IoC container** which is a parent interface of **ApplicationContext.** BeanFactory does not support Annotation-based configuration whereas ApplicationContext does.

Example:

1. class Student -> ( name , age )
2. spring.xml (placed in src/main/resources)

demo-

<?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 id="student1" class="com.woletrs.beanDemo.pojos.Student">

<constructor-arg name="name" value="Ram" />

<constructor-arg name="age" value="18" />

</bean>

</beans>

1. Main Class –

@SpringBootApplication

public class BeanDemoApplication {

public static void main(String[] args) {

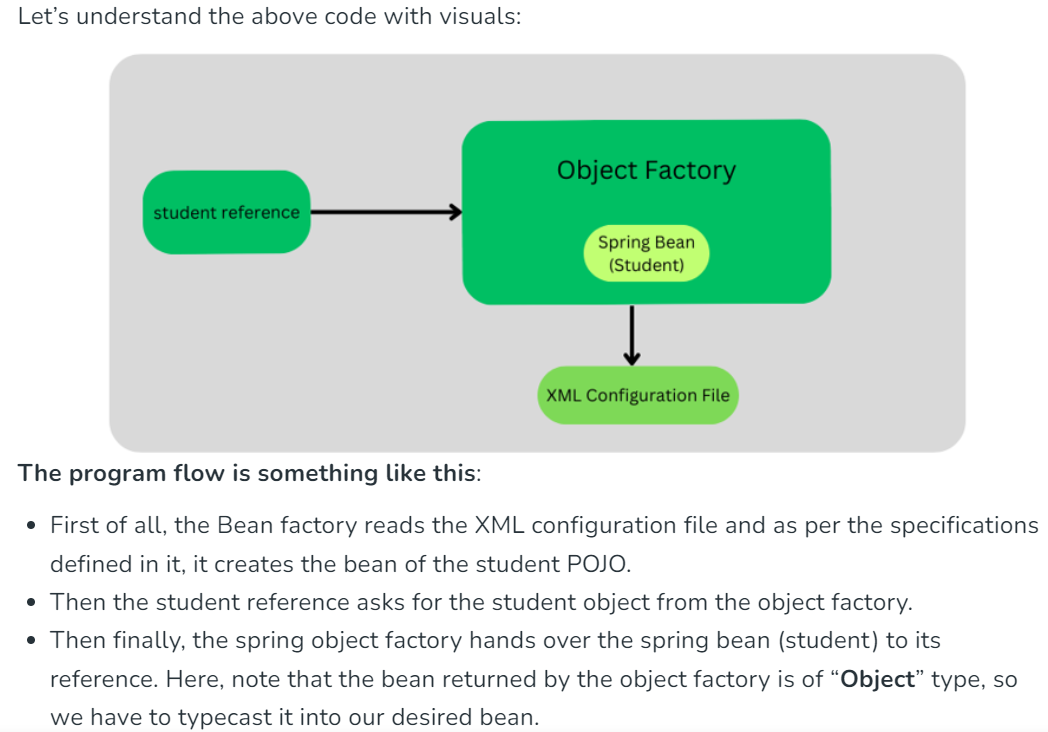
BeanFactory factory = new ClassPathXmlApplicationContext("spring.xml");

Student student = (Student) factory.getBean("student1");

System.out.println(student);

}

}



# II. Spring – ApplicationContext:

**ApplicationContext** belongs to the **Spring**framework. **Spring IoC contain**er is responsible for instantiating, wiring, configuring, and managing the entire life cycle of beans or objects.BeanFactory and ApplicationContext represent the Spring IoC Containers. **ApplicationContext is the sub-interface of BeanFactory**. It is used when we are creating an enterprise-level application or web application. ApplicationContext is the superset of BeanFactory, whatever features provided by BeanFactory are also provided by ApplicationContext.

**ApplicationContext Implementation Classes-**

There are different types of Application containers provided by Spring for different requirements as listed below which later onwards are described alongside with declaration, at lastly providing an example to get through the implementation part with the pictorial aids. Containers are as follows:

1. AnnotationConfigApplicationContext container
2. AnnotationConfigWebApplicationContext
3. XmlWebApplicationContext
4. FileSystemXmlApplicationContext
5. ClassPathXmlApplicationContext

**AnnotationConfigApplicationContext container Example:**

Create**Student**Class (id,name) under com.woletrs.AnnotationConfigApplicationContext.pojos

Create AppConfig class under com.woletrs.AnnotationConfigApplicationContext.config

Code:

@Configuration

public class AppConfig {

@Bean

public Student student()

{

return new Student(1,"Ram");

}

}

Inside MainMethod of MainClass:

Code:

@SpringBootApplication

public class AnnotationConfigApplicationContextApplication {

public static void main(String[] args) {

// ApplicationContext context = SpringApplication.run(AnnotationConfigApplicationContextApplication.class, args);

// Student student = context.getBean(Student.class);

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Student student = (Student) context.getBean(Student.class);

System.out.println(student);

}

}

Difference Between BeanFactory and ApplicationContext:

**Dependency Injection:**Dependency Injection is a design pattern that allows the spring container to ‘inject’ objects into other objects or dependencies. In simple words, the control of creating objects and managing the spring components is taken care of by the Spring containers.

### **Spring Containers**

Spring Framework provides two of the most fundamental and important packages, they are the **org.springframework.beans** and **org.springframework.context** packages. Code in these packages provides the basis for Spring’s **Inversion of Control/Dependency Injection** features. Spring containers are responsible for creating bean objects and injecting them into the classes. The two containers are namely,

1. **BeanFactory(I)** – Available in org.springframework.beans.factory package.
2. **ApplicationContext(I)**– Available in org.springframework.context package.

### ***BeanFactory Interface****:*

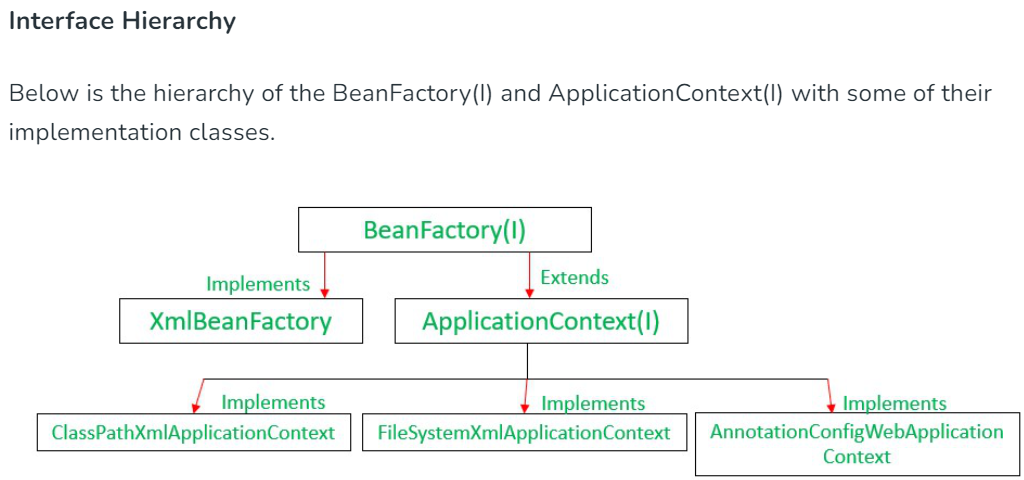
This is the root interface for accessing a Spring bean container. It is the actual container that instantiates, configures, and manages a number of beans. These beans collaborate with one another and thus have dependencies between themselves. These dependencies are reflected in the configuration data used by the BeanFactory. This interface is implemented by the objects that hold a number of bean definitions, each uniquely identified by a String name

***ApplicationContext Interface:***

This interface is designed on top of the BeanFactory interface. The ApplicationContext interface is the advanced container that enhances BeanFactory functionality in a more framework-oriented style. While the BeanFactory provides basic functionality for managing and manipulating beans, often in a programmatic way, the ApplicationContext provides extra functionality like MessageSource, Access to resources, Event propagation to beans, Loading of multiple (hierarchical) contexts etc. There are so many implementation classes that can be used such as **ClassPathXmlApplicationContext**, **FileSystemXmlApplicationContext**, **AnnotationConfigWebApplicationContext** etc.

**Example code:**

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



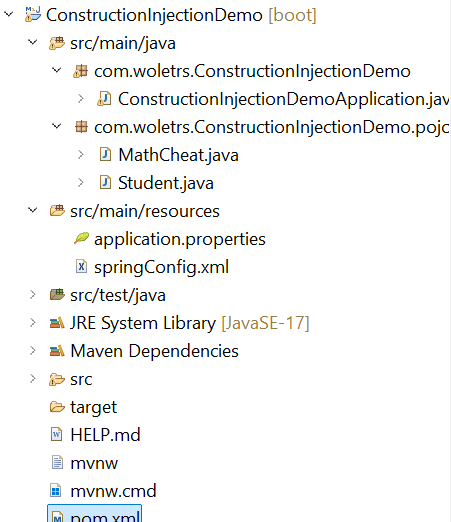
Below is the difference table of Spring BeanFactory and ApplicationContext.

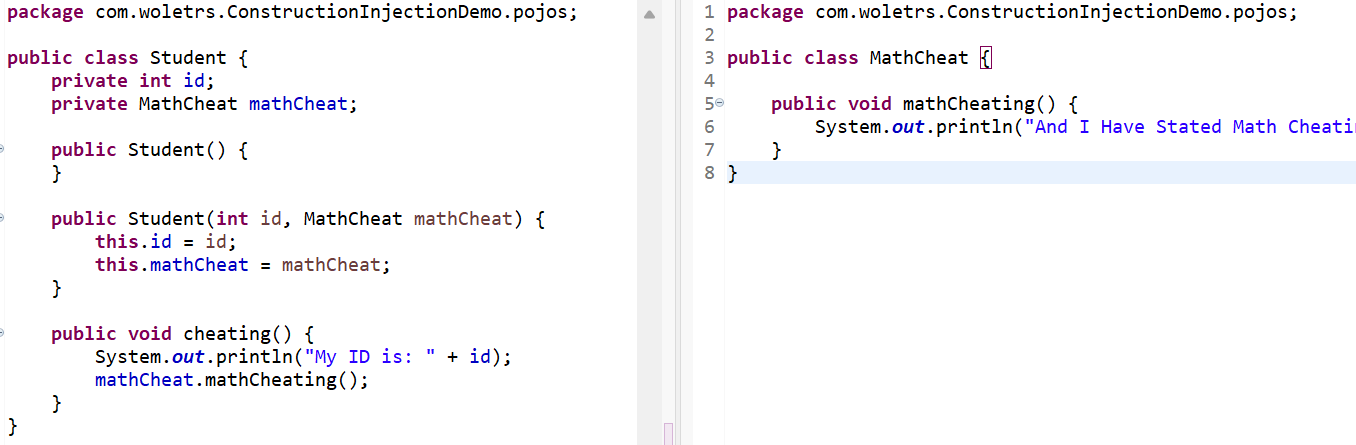
| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| **Definition** | Fundamental container providing basic functionality for managing beans. | Advanced container extending BeanFactory with additional features. |
| **Usage** | Suitable for building standalone applications. | Suitable for building web applications, integrating with AOP modules, ORM, and distributed applications. |
| **Bean Scopes Supported** | Supports only Singleton and Prototype bean scopes. | Supports all types of bean scopes, including Singleton, Prototype, Request, Session, etc. |
| **Annotation Support** | Does not support annotations; requires configuration in XML files. | Supports annotation-based configuration for bean autowiring. |
| **Internationalization** | Does not provide internationalization (i18n) functionality. | Extends MessageSource interface to provide internationalization (i18n) functionality. |
| **Event Handling** | Does not support event publication. | Supports event handling via the ApplicationEvent class and ApplicationListener interface. |
| **Bean Post Processing** | Requires manual registration of BeanPostProcessors and BeanFactoryPostProcessors. | Automatically registers BeanFactoryPostProcessor and BeanPostProcessor at startup. |
| **Initialization** | Creates bean objects on demand using lazy initialization. | Loads all beans and creates objects at startup using eager initialization. |
| **Resource Usage** | Provides basic features requiring less memory, suitable for memory-critical standalone applications. | Provides basic and advanced features, suitable for enterprise applications, requiring more memory. |

### **Constructor Injection**

In Constructor Injection, the Dependency Injection will be injected with the help of constructors. Now to set the Dependency Injection as Constructor Dependency Injection in bean, it is done through the bean-configuration file. For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

Example:

SetUp -

Student Class: MathCheat Class:

springConfig.xml 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 id=*"MathCheatRef"* class=*"com.woletrs.ConstructionInjectionDemo.pojos.MathCheat"* />

<bean id=*"Student1"* class=*"com.woletrs.ConstructionInjectionDemo.pojos.Student"*>

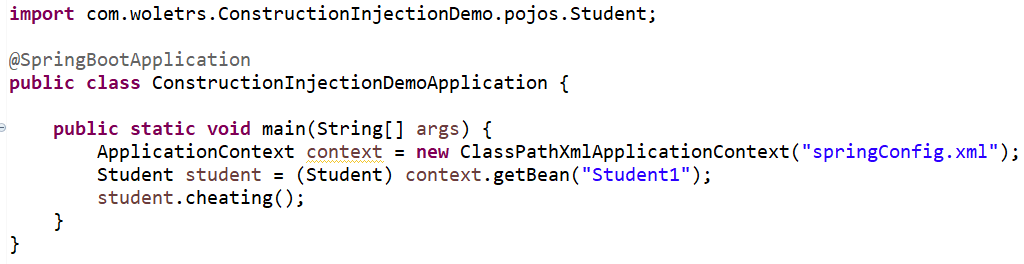
<constructor-arg name=*"id"* value=*"100"* />

<constructor-arg name=*"mathCheat"* ref=*"MathCheatRef"* />

</bean>

</beans>

MainApp:



Output:

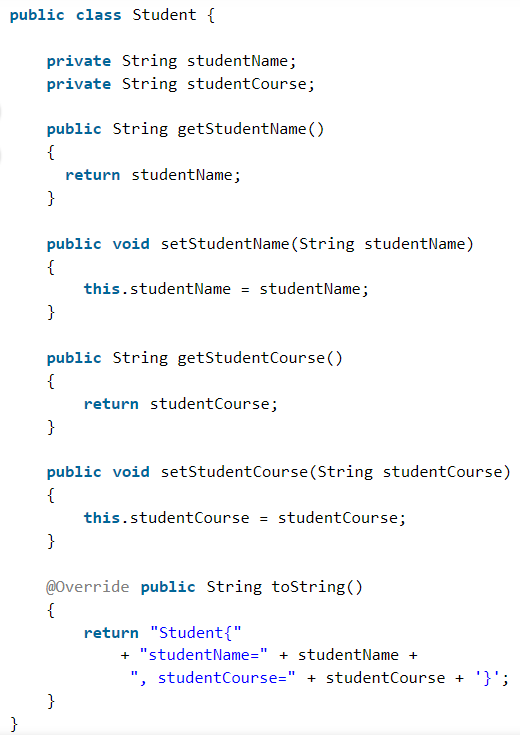


# Spring – Setter Injection.

This is done by the container calling setter methods on beans after invoking a no-argument constructor or no-argument static factory method to instantiate the bean. We need to use **<property>** sub-element of <bean> tag for setter injection.

Let’s see an example to inject dependency by the setter method.

1. **Employee.java (POJO class)**
2. **config.xml**
3. **Main.java**



springConfig.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"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<**bean** class="com.springframework.Student" name="stud">

       <**property** name="studentName">

               <**value**> John </**value**>

       <**property**/>

       <**property** name="studentCourse">

               <**value**> Spring Framework </**value**>

          <**property**/>

    </**bean**>

</beans>

Main Class:

**public** **class** GFG {

**public** **static** **void** main(String[] args)

    {

        ApplicationContext context = **new** ClassPathXmlApplicationCotenxt("config.xml");

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

        System.out.println(student);

    }

}

Output:

Student{ studentName= John , studentCourse= Spring Framework }

# Setter Injection with Dependent Object:

If there exists a relationship between the classes of our spring application, then we create the instance of the dependent object also called the **contained object**. After creating the instance of the dependent object we pass it as an argument of the main class container.

Example: If an employee has an address, then the instance of the Address class is the dependent object and will be contained in the Employee class.

**Implementation:** The following example demonstrates a setter injection with a dependent object.

**A.**Employee.java - Each employee has the following properties:

Name, Employee ID, Department, ***Address*** (Dependent object)

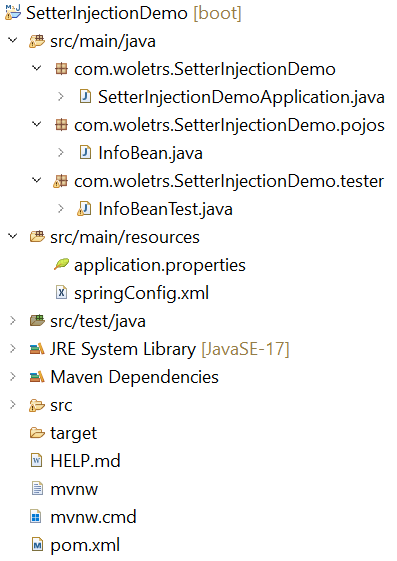
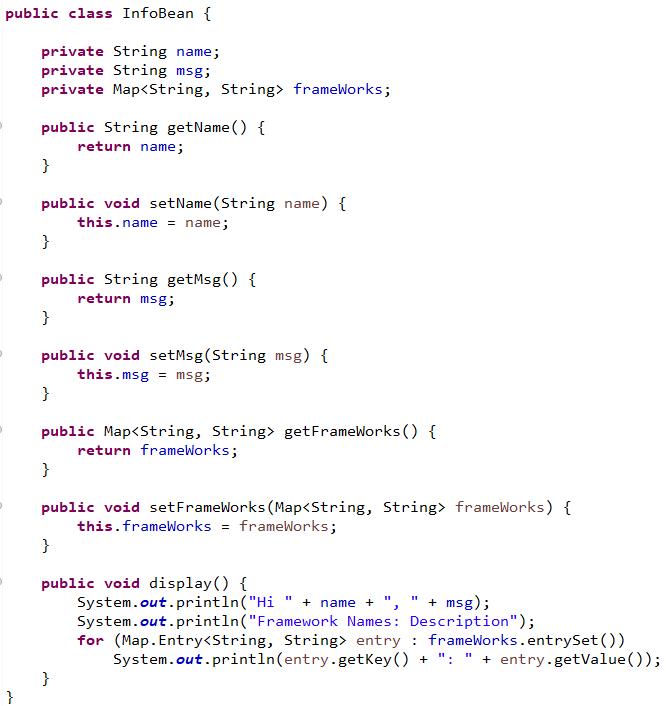
**B.**Address.java- Address has the following properties:

 House Number, Pincode, State, Country

# Setter Injection with Map:

Through setter injection using <property> tag, we can provide values like Strings, Primitives, Collections, etc. based on our requirement. In this example, we will see setter injection using Map. While we are using the map, we need to use **<map>** and **<entry>** tags to set the values to it.

Example-



InfoBean Class:

springConfig.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"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"InfoBean1"*

class=*"com.woletrs.SetterInjectionDemo.pojos.InfoBean"*>

<property name=*"name"* value=*"Ratnakar"*></property>

<property name=*"msg"* value=*"Welcome to SpringDemo"*></property>

<property name=*"frameWorks"*>

<map>

<entry key=*"Spring"*

value=*"Spring is an application framework used to develop Java Enterprise applications and designed on concept called Dependency injection."*></entry>

<entry key=*"Hibernate"*

value=*"Hibernate is an object-relational mapping (ORM) framework that works with relational databases to manage the data."*></entry>

</map>

</property>

</bean>

</beans>

Tester Class:

**public** **class** InfoBeanTest {

**public** **static** **void** main(String[] args) {

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

InfoBean infoBean = (InfoBean) context.getBean("InfoBean1");

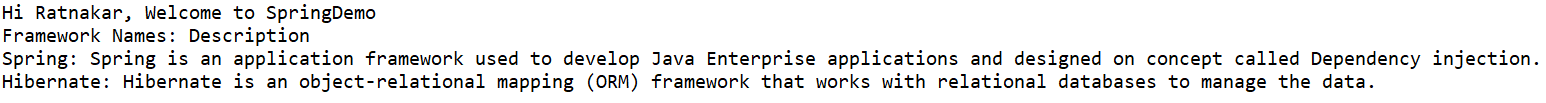
infoBean.display();

}

}

Output:

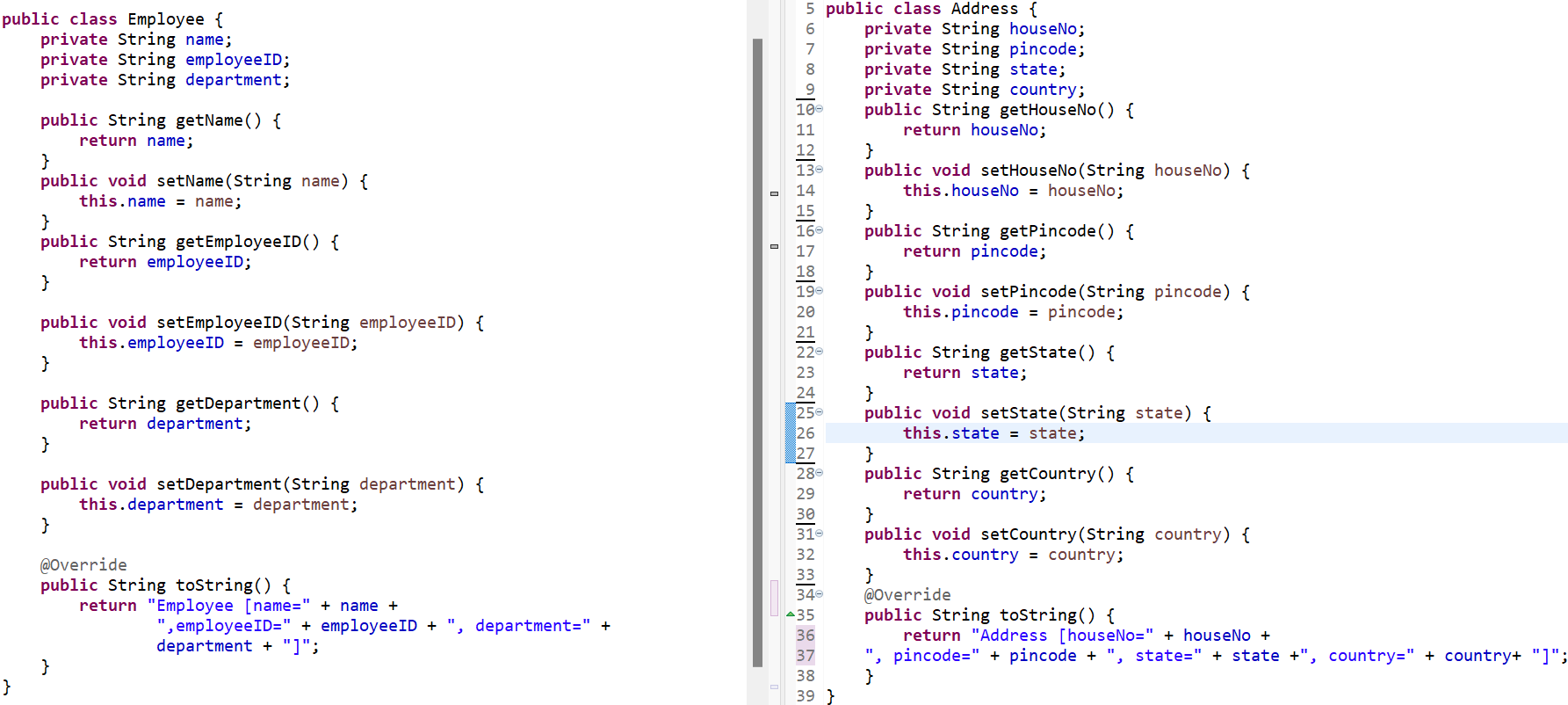
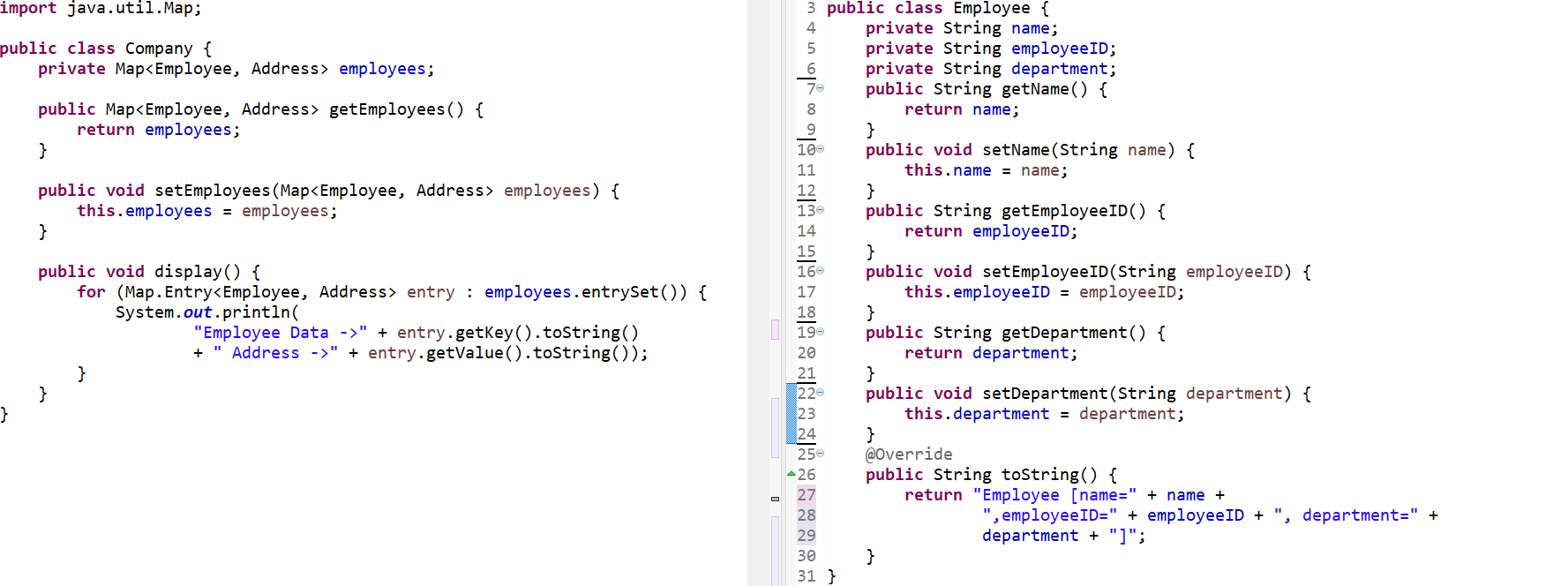
* Run the test class as a Java project.
* The Spring container will create the ‘InfoBean’ object and it sets all the property values and make the bean object available to the developer.
* Using the bean object, it calls the display() method and prints the below output in the console.



# Setter Injection with Non-String Map:

The map will have both key and value as non-strings. Key will be Employee which has the following fields: Name, Employee ID, Department

Value will be Address which has the following parameters: House No., Pincode, State, Country



springConfig.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"*

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

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

<bean id=*"employee1"* class=*"com.woletrs.SetterInject\_NSM.pojos.Employee"*>

<property name=*"name"* value=*"Ratnakar"*></property>

<property name=*"employeeID"* value=*"100"*></property>

<property name=*"department"* value=*"Computer Engineering"*></property>

</bean>

<bean id=*"address1"* class=*"com.woletrs.SetterInject\_NSM.pojos.Address"*>

<property name=*"houseNo"* value=*"RNP500"*></property>

<property name=*"pincode"* value=*"415101"*></property>

<property name=*"state"* value=*"Maharashtra"*></property>

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

</bean>

<bean id=*"company1"* class=*"com.woletrs.SetterInject\_NSM.pojos.Company"*>

<property name=*"employees"*>

<map>

<entry key-ref=*"employee1"* value-ref=*"address1"*></entry>

</map>

</property>

</bean>

</beans>

Output:

J2EE- Java Enterprise Edition

POJO- Plain Old Java Objects

EJB- Enterprise Java Beans

ORM – Object-Relational Mapping

JPA- Java Persistence API

JDO- Java Data Objects

IMP LINK: [Spring Tutorial (tutorialspoint.com)](https://www.tutorialspoint.com/spring/index.htm)

**\*\* Simple Explanation: Dependency Injection \*\***

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods.

Following two important points are to be noted about the main program −

* The first step is to create an **application context** where we used framework API **ClassPathXmlApplicationContext()**. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
* The second step is used to get the required bean using **getBean()** method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

**Bean Life Cycle:**

Flow is- First Object creation -> Properties set -> public void init method () -> Intermediate work -> public void destroy method () -> Object Destroyed.

1. Xml Based Configuration of Bean life cycle:

( Init-method, destroy method )

[Note: to call destroy () method, you have to mentioned registerShutdownHook() method from AbstarctApplicationContext Interface. ]

1. Interface Based configuration of Bean life cycle.

( Initializing Bean -> Overridable method: afterPropertiesSet(), Disposable Bean -> Overridable method: destroy() )

1. Annotation Based configuration of Bean life cycle.

@PostConstruct @PreDestroy also they are now deprecated from java EE hence you have to add dependency of “annotation-api” in pom.xml.

Also add <context:annotation-config/> inside config.xml file to enable all annotations but if you want to enable only selected annotations then you have to create a <bean class=”org.springframework.context.annotation.CommonnAnnotationBeanPostProcessor” which enable only specific annotations that are beingbused.

Example:

..

..

@PostConstruct

Public void start() {…..}

@PreDestroy

Public void end(){..}

**Autowiring:** It is something in which, spring container automatically inject the dependency into the bean. OR Feature of spring framework in which spring container inject the dependencies automatically.

In Short- Spring container automatically finding the dependencies and injecting them into the appropriate bean.

**Why?** autowiring do not support primitive types-> It's because Autowiring is just an alternative for referencing your existing beans in ApplicationContext. It expects a bean or a class, primitive is not a class and it differs from Object. Which is why you can't Autowire primitive types. You can use wrapper classes of the primitive types such as Integer, Double etc... to be able to use Autowiring for such types because you are now referring to a class.

Manual 🡪 <Bean id=”**refA**” class=”…”> …… <bean/>

<Bean id=”BeanB” class=”…”>

<properties name=”…” ref-bean=”**refA**” />

<bean/>

Two ways:

1. XML:

Autowiring Modes ->

* 1. No -It is by default there.
  2. byName: Spring container search for the name of bean in configurations and if find then inject that dependency in required space. (Uses Setter DI)
  3. byType : Spring container search for the Typeof bean in configurations and if find then inject that dependency in required bean. It will not worry about the name of the bean just check the Type (Eg. Address) of the bean,
  4. constructor: same as byName but instead of setter DI it uses Constructor DI.

1. Annotation:

Using @AutoWired:

[ Note: add this inside the bean:

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.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"  
  
 <context:annotation-config> </context:annotation-config>

1. On property (uses byType):
2. On setter method (uses setter injection):

So it will work like byName and setter DI.

1. On constructor (uses constructor injection):
2. So it will work like constructor DI.

Advantages of autowiring:

1. Less code
2. Automatic

Disadvantage of autowirng:

1. No control of programmer
2. It cant be used for primitive data types and Strings, works only for objects.