Inversion of Control and Dependency Injection

The principle/Feature where instead of client doing activity of managing the objects

Control is taken over by the container/Framework is called inversion of control.

Spring is using DI flavor of inversion of control.

Bean Definition Configuration File

Major configuration feature

1. Making association between bean objects using ref element
2. Mapping collection properties by different collection properties
3. Importing configuration files into master xml file

   <beans>

   <import resource="emp.xml"/>---contains employee pojo related configuration details

   <import resource="address.xml"/>--contains address pojo related configuration details

   </beans>

   These various xml files are combined together to form master xml

   This master xml will be used to create beanfactory by passing it as resource

   Thus we will achieve modularization.

Example Program

Invoice.java

```
package com.spring.collection_mapping;

import java.util.Set;

public class Invoice {
	private String invoiceId;
	private int totalAmount;
	private Set <Products>products;
	public String getInvoiceId() {
		return invoiceId;
	}
	public void setInvoiceId(String invoiceId) {
		this.invoiceId = invoiceId;
	}
	public int getTotalAmount() {
		return totalAmount;
	}
```

```
	public void setTotalAmount(int totalAmount) {
		this.totalAmount = totalAmount;
	}
	public Set<Products> getProducts() {
		return products;
	}
	public void setProducts(Set<Products> products) {
		this.products = products;
	}


}
```

Products.java

```
package com.spring.collection_mapping;

public class Products {
	private String productId;
	private String productName;
	private int productPrice;
	public String getProductId() {
		return productId;
	}
	public void setProductId(String productId) {
		this.productId = productId;
	}
	public String getProductName() {
		return productName;
	}
	public void setProductName(String productName) {
		this.productName = productName;
	}
	public int getProductPrice() {
		return productPrice;
	}
	public void setProductPrice(int productPrice) {
		this.productPrice = productPrice;
	}



}
```

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

<bean id="tableBean" class="com.spring.collection_mapping.Products">
	<property name="productId" value="P001"></property>
	<property name="productName" value="Table"></property>
	<property name="productPrice" value="1000"></property>
</bean>
```

```
<bean id="chairBean" class="com.spring.collection_mapping.Products">
	<property name="productId" value="P002"></property>
	<property name="productName" value="Chair"></property>
	<property name="productPrice" value="500"></property>
</bean>
<bean id="invoiceBean" class="com.spring.collection_mapping.Invoice">
<property name="invoiceId" value="INV100"></property>
<property name="products">
	<set>
	<ref bean="tableBean"/>
	<ref bean="chairBean"/>
	</set>

</property>


</bean>

</beans>
```

Note : By accessing(By calling) setter method of pojo it(property of bean in xml) will. So this way of injecting dependencies between beans and property is basically called dependency injection. Here we use setter method to inject dependency. This is one type of DI.it will call setter method and will inject the values.This technique is called setter injection.

CollectionClient.java

```
package com.spring.collection_mapping;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

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

public class CollectionClient {

	public static void main(String[] args) {
		Resource xmlResource= new FileSystemResource("Resource.xml");
		BeanFactory factory=new XmlBeanFactory(xmlResource);
		Invoice invoice=(Invoice)factory.getBean("invoiceBean");
		Products product1=(Products)factory.getBean("tableBean");
		Products product2=(Products)factory.getBean("chairBean");
		Set<Products> products=new HashSet<Products>();
		products.add(product1);
		products.add(product2);
		int totalPrice=product1.getProductPrice()+product2.getProductPrice();
		invoice.setTotalAmount(totalPrice);
		invoice.setProducts(products);
		Iterator itr=products.iterator();
```

```
            System.out.println("Please check your system generated invoice");
            System.out.println("-------------------------------------------------
--");
            System.out.println("Invoice ID : "+invoice.getInvoiceId());
            while(itr.hasNext())
            {
                  Products product=(Products)itr.next();
                  System.out.println("Product Name : "+product.getProductName()+"
      Product Price :"+product.getProductPrice());
            }


            System.out.println("Total Amount : "+invoice.getTotalAmount());
      }

}
```

Component wiring means making association betwwn various objects.

Instead of client program managing the objects framework (spring container) manages all these. Means control is reversed to container called IOC
Thumb Rule for DI

DI is one flavor of IOC. DI is done in two ways one is setter injection as we saw . Another is constructer injection.

Setter Injection : In the configuration file if the element called “property” is used then it is clear example of setter injection. It looks at the property and calls its appropriate setter method to inject/set the values.

Constructor Injection : Injection via constructor argument

Note : Constructor is called only once when object is initialized. Later we cannot change this but using setter injection we can change this as we can call setter method any number of times

If you are using constructor injection then must create parameterized constructor in bean class otherwise following error is encountered. In setter injection constructor is not required.

Error creating bean with name 'address' defined in class path resource [com/config/SpringContext.xml]: Could not resolve matching constructor (hint: specify index/type/name arguments for simple parameters to avoid type ambiguities)

Note : If bean class contains 3 members and you want only two members then pass only 2 parameters in in parameterized constructor in bean class otherwise runtime exception will be thrown.

Ways of Constructor Injection ;

Name and Value

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

    <constructor-arg name="doorNumber" value="101"></constructor-arg>
    <constructor-arg name="city" value="chennai"></constructor-arg>
    <constructor-arg name="state" value="TN"></constructor-arg>
                                          </bean>
```

Index, Data type and value

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

    <constructor-arg type="Java.lang.int" index="0" value="101"></constructor-arg>
   <constructor-arg type="Java.lang.String" index="1" value="chhh"></constructor-arg>
   <constructor-arg type="Java.lang.String" index="2" value="tr"></constructor-arg>

                                          </bean >
```

Suppose for a bean class in context file contains both injections then what will happen?

Consider the below scenario-

```
<bean id="address" class="com.beans.Address">
    <property name="doorNumber" value="101"></property>
    <property name="city" value="Chennai"></property>
    <property name="state" value="Tamilnadu"></property>

    </bean>

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

     <constructor-arg name="doorNumber" value="101"></constructor-arg>
    <constructor-arg name="city" value="chennai"></constructor-arg>
    <constructor-arg name="state" value="TN"></constructor-arg>
                                          </bean>
```

Ans : Runtime exception : BeanDefinitionParsingException: Configuration problem: Bean name 'address' is already used in this <beans> element

What happens if a spring-context file contains an abstract class

```
<bean id="department" class="com.beans.Department"></bean>
```

Ans : As we know abstract class cannot be instantiated so runtime exception is encountered.

If we modify it like

```
<bean id="department" class="com.beans.Department" abstract="true"></bean>
```

So here we are telling container it is an abstract class Now program will run without exception

How to read properties file value in spring-context.xml file?

Suppose your properties file contains following data

```
doorNumber=52
city=Hyderabad
state=Telangana
```

then make following changes in spring-context.xml file

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

    <property name="locations" value="classpath:com/resources/info.properties"/>
    </bean>



      <bean id="address" class="com.beans.Address">
    <property name="doorNumber" value="${doorNumber}"></property>
    <property name="city" value="${city}"></property>
    <property name="state" value="${state}"></property>

    </bean>
```

If you change property name something else (say loc) than locations you will found runtime error invalid property loc.

Concept of AutoWire :

By default autowire is switched off in spring.

It can be in two ways :

Auto-wiring= “byType” : look for a bean whose type(class) is same as the property.

Auto-wiring=”byName” : look for the id or name of the bean which is same as the property

Example of byType and byName

Employee.java

```
package beans;

public class Employee {
	private int empId;
	private String name;
	private Address homeAddress;
	private Address officeAddress;

	public Address getHomeAddress() {
		return homeAddress;
	}


	public void setHomeAddress(Address homeAddress) {
		this.homeAddress = homeAddress;
	}


	public Address getOfficeAddress() {
		return officeAddress;
	}


	public void setOfficeAddress(Address officeAddress) {
		this.officeAddress = officeAddress;
	}


	public Employee() {
		super();
		System.out.println("Employee Constructor");
	}


	public int getEmpId() {
		return empId;
	}
	public Employee(int empId, String name) {
		super();
		this.empId = empId;
		this.name = name;
	}
	public void setEmpId(int empId) {
		this.empId = empId;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
```

Configuration.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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

       <bean id="employee" class="beans.Employee" autowire="byName">
       <!--If we want only one variable of Address then we can use byType then it will
search for type rather than name or id as it is searching
       in byName -->
       <property name="empId" value="330139"></property>

       <property name="name" value="Sanjay Rai"></property>
       </bean>

       <bean id="officeAddress" class="beans.Address" >
       <property name="doorNumber" value="101"></property>
       <property name="city" value="Chennai"></property>
       <property name="state" value="Tamilnadu"></property>

       </bean>

       <bean id="homeAddress" class="beans.Address" parent="officeAddress">
        <property name="doorNumber" value="201"></property>
       <property name="city" value="Patna"></property>
       <property name="state" value="Bihar"></property>


       </bean>

       <bean id="homeAddress" class="beans.Address" parent="officeAddress"><!-- If we
just write this in homeAddress, officeAddress will be copied -->
       </bean>

</beans>
<!-- If we use autowire=byType and that bean has two instances officeAddress and
homeAddress and then ambiguity exception is thrown.
To overcome this please use autowire byName -->
```

Address.java

```
package beans;

public class Address {
	private int doorNumber;
	private String city;
	private String state;

//	public Address(int doorNumber, String city, String state) {
//		super();
//		this.doorNumber = doorNumber;
//		this.city = city;
//		this.state = state;
//	}
//

	public int getDoorNumber() {
		return doorNumber;
	}
	public void setDoorNumber(int doorNumber) {
		this.doorNumber = doorNumber;
	}
	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;
	}
```

Main.java

```
package beans;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

	public static void main(String args[])
	{
		ApplicationContext context =new
ClassPathXmlApplicationContext("beans/configuration.xml");
		Employee emp=(Employee) context.getBean("employee");
		System.out.println(emp.getEmpId());
		System.out.println(emp.getName());
		System.out.println(emp.getHomeAddress().getCity());

	}
```

Why application context over bean factory?

ApplicationContext Container is advanced than Beanfactory Container...

1) BeanFactory Container is basic container, it can only create objects and inject Dependencies.But we can not attach other services like security, transaction ,messaging etc. To provied all the services we have to use ApplicationContext Container.

2) BeanFactory Container doesn't support the feature of AutoScanning , but ApplicationContext Container supports.

3) Beanfactory Container will not create a bean object upto the request time.It means Beanfactory Container loads beans laziely. While ApplicationContext Container creates objects of Singleton bean at the time of loading only.It means there is early loading.

4) Beanfactory Container support only two scopes (singleton & prototype) of the beans. But ApplicationContext Container supports all the beans scope.