Spring - Dependency Injection

Every java based application has a few objects that work together to present what the end-user sees as a working application. 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 doing unit testing.

Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.

Dependency Injection is possible with adding a Constructor Argument and with setter method.

Spring Constructor-based Dependency Injection

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

Demo:

```
package com.h2kinfosys;

public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

So TextEditor has SpellChecker dependency:

```
package com.h2kinfosys;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }
}
```

We can set this DI using below configuration:

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor">
      <constructor-arg ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>
</beans>
```

Following is the content of the **MainApp.java** file:

```
package com.h2kinfosys;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
      TextEditor te = (TextEditor) context.getBean("textEditor");
      te.spellCheck();
   }
}
```

There may be a ambiguity exist while passing arguments to the constructor in case there are more than one parameters. To resolve this ambiguity, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor.

Let us check a case where we pass different types to the constructor. Consider the following class:

```
package x.y;
public class Foo {
   public Foo(int year, String name) {
      // ...
   }
}
```

The container can also use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute. For example:

```
<beans>
   <bean id="exampleBean" class="examples.ExampleBean">
      <constructor-arg type="int" value="2001"/>
      <constructor-arg type="java.lang.String" value="Zara"/>
   </bean>
</beans>
```

Finally and the best way to pass constructor arguments, use the index attribute to specify explicitly the index of constructor arguments. Here the index is 0 based. For example:

```
<beans>
   <bean id="exampleBean" class="examples.ExampleBean">
      <constructor-arg index="0" value="2001"/>
      <constructor-arg index="1" value="Zara"/>
   </bean>
</beans>
```

Spring Setter-based Dependency Injection

Following is the configuration file Beans.xml which has configuration for the setter-based injection: .

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor">
      <property name="spellChecker" ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>
</beans>
```

Following is the content of the MainApp.java file:

```
package com.h2kinfosys;


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


public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
      TextEditor te = (TextEditor) context.getBean("textEditor");
      te.spellCheck();
```

```
  }
}
```

XML Configuration using p-namespace:

If you have many setter methods then it is convenient to use p-namespace in the XML configuration file.

Above XML configuration can be re-written in a cleaner way using p-namespace as follows:

```
<?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="john-classic" class="com.example.Person"
      p:name="John Doe"
      p:spouse-ref="jane"/>
   </bean>

   <bean name="jane" class="com.example.Person"
      p:name="John Doe"/>
   </bean>
</beans>
```

Spring - Injecting Inner Beans

a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean and it is shown below.

```
<?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">
```

```
  <bean id="outerBean" class="...">
    <property name="target">
      <bean id="innerBean" class="..."/>
    </property>
  </bean>
</beans>
```

Following is the configuration file Beans.xml which has configuration for the setter-based injection but using inner 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-3.0.xsd">

  <!-- Definition for textEditor bean using inner bean -->
  <bean id="textEditor" class="com.h2kinfosys.TextEditor">
    <property name="spellChecker">
      <bean id="spellChecker" class="com.h2kinfosys.SpellChecker"/>
     </property>
  </bean>
</beans>
```

Spring - Injecting Collection

Element	**Description**
<list>	This helps in wiring ie injecting a list of values, allowing duplicates.
<set>	This helps in wiring a set of values but without any duplicates.
<map>	This can be used to inject a collection of name-value pairs where name and value can be of any type.
<props>	This can be used to inject a collection of name-value pairs where the name and value are both Strings.

```
import java.util.*;
```

```

public class JavaCollection {
   List addressList;
   Set  addressSet;
   Map  addressMap;
   Properties addressProp;

   // a setter method to set List
   public void setAddressList(List addressList) {
      this.addressList = addressList;
   }

   // prints and returns all the elements of the list.
   public List getAddressList() {
      System.out.println("List Elements :"  + addressList);
      return addressList;
   }

   // a setter method to set Set
   public void setAddressSet(Set addressSet) {
      this.addressSet = addressSet;
   }

   // prints and returns all the elements of the Set.
   public Set getAddressSet() {
      System.out.println("Set Elements :"  + addressSet);
      return addressSet;
   }

   // a setter method to set Map
   public void setAddressMap(Map addressMap) {
      this.addressMap = addressMap;
   }

   // prints and returns all the elements of the Map.
   public Map getAddressMap() {
      System.out.println("Map Elements :"  + addressMap);
      return addressMap;
   }

   // a setter method to set Property
   public void setAddressProp(Properties addressProp) {
      this.addressProp = addressProp;
   }

   // prints and returns all the elements of the Property.
   public Properties getAddressProp() {
      System.out.println("Property Elements :"  + addressProp);
      return addressProp;
   }
}
```

Following is the configuration file **Beans.xml** which has configuration for all the type of collections:

```
<?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">

   <!-- Definition for javaCollection -->
   <bean id="javaCollection" class="com.h2kinfosys.JavaCollection">

      <!-- results in a setAddressList(java.util.List) call -->
```

```
<property name="addressList">
  <list>
    <value>INDIA</value>
    <value>Pakistan</value>
    <value>USA</value>
    <value>USA</value>
  </list>
</property>

<!-- results in a setAddressSet(java.util.Set) call -->
<property name="addressSet">
  <set>
    <value>INDIA</value>
    <value>Pakistan</value>
    <value>USA</value>
    <value>USA</value>
 </set>
</property>

<!-- results in a setAddressMap(java.util.Map) call -->
<property name="addressMap">
  <map>
    <entry key="1" value="INDIA"/>
    <entry key="2" value="Pakistan"/>
    <entry key="3" value="USA"/>
    <entry key="4" value="USA"/>
  </map>
</property>

<!-- results in a setAddressProp(java.util.Properties) call -->
<property name="addressProp">
  <props>
    <prop key="one">INDIA</prop>
    <prop key="two">Pakistan</prop>
```

```
            <prop key="three">USA</prop>
            <prop key="four">USA</prop>
         </props>
      </property>
   </bean>
</beans>
```

Injecting null and empty string values

If you need to pass an empty string as a value then you can pass it as follows:

```
<bean id="..." class="exampleBean">
   <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail("")

If you need to pass an NULL value then you can pass it as follows:

```
<bean id="..." class="exampleBean">
   <property name="email"><null/></property>
</bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail(null)

Spring - Beans Auto-Wiring

The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

Autowiring Modes:

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the **autowire** attribute of the <bean/> element to specify autowire mode for a bean definition.

Mode	**Description**
no	This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is

	what you already have seen in Dependency Injection chapter.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byName*in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
byType	Autowiring by property datatype. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byType*in the XML configuration file. It then tries to match and wire a property if its **type** matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown.
Constructor	Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Spring first tries to wire using autowire by *constructor*, if it does not work, Spring tries to autowire by *byType*.

Spring Autowiring 'byName'

This mode specifies autowiring by property name. Spring container looks at the beans on which *auto-wire* attribute is set to *byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions

Following is the configuration file **Beans.xml** in normal condition:

```
<?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">


   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor">
      <property name="spellChecker" ref="spellChecker" />
```

```
     <property name="name" value="Generic Text Editor" />
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>

</beans>
```

But if you are going to use autowiring 'byName', then your XML configuration file will become as follows:

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor" autowire="byName">
     <property name="name" value="Generic Text Editor" />
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>

</beans>
```

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

```
Inside SpellChecker constructor.
Inside checkSpelling.
```

Spring Autowiring 'byType'

Following is the configuration file **Beans.xml** in normal condition:

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor">
      <property name="spellChecker" ref="spellChecker" />
      <property name="name" value="Generic Text Editor" />
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>
</beans>
```

But if you are going to use autowiring 'byType', then your XML configuration file will become as follows:

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor"
      autowire="byType">
      <property name="name" value="Generic Text Editor" />
```

```
  </bean>


  <!-- Definition for spellChecker bean -->
  <bean id="SpellChecker" class="com.h2kinfosys.SpellChecker">
  </bean>
</beans>
```

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

```
Inside SpellChecker constructor.
Inside checkSpelling.
```

Spring Autowiring by Constructor

This mode is very similar to *byType*, but it applies to constructor arguments. Spring container looks at the beans on which *autowire* attribute is set to *constructor* in the XML configuration file. It then tries to match and wire its constructor's argument with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

Following is the configuration file **Beans.xml** in normal condition:

```
<?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">

  <!-- Definition for textEditor bean -->
  <bean id="textEditor" class="com.h2kinfosys.TextEditor">
    <constructor-arg  ref="spellChecker" />
    <constructor-arg  value="Generic Text Editor"/>
  </bean>

  <!-- Definition for spellChecker bean -->
  <bean id="spellChecker" class="com.h2kinfosys.SpellChecker">
  </bean>

</beans>
```

But if you are going to use autowiring 'by constructor', then your XML configuration file will become as follows:

```
<?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">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.h2kinfosys.TextEditor"
      autowire="constructor">
      <constructor-arg value="Generic Text Editor"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="SpellChecker" class="com.h2kinfosys.SpellChecker">
   </bean>

</beans>
```

Limitations with autowiring:

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

Limitations	Description
Overriding possibility	You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.

Primitive data types	You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
Confusing nature	Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring.