Setter Injection with Collection Example

We can inject collection values by setter method in spring framework. There can be used three elements inside the **property** element.

It can be:

1. **list**
2. **set**
3. **map**

Each collection can have string based and non-string based values.
In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages:

1. **Question.java**
2. **Beans.xml**
3. **Test.java**

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

Step 1: Create a Maven Project

group id : com.spring.SI.Collection

Artifact id: Spring-SI-Collection

Step 2: Open pom.xml and add the dependency of spring-context

Step 3: Create a package com.sping.SI.Collection

Step 4: Inside the package - Create Question.java and Test.java

Step 5: Create the Beans.xml

Question.java

This class contains three properties with setters and getters and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
import java.util.Iterator;
import java.util.List;

public class Question {
private int id;
private String name;
private List<String> answers;

//setters and getters

public void displayInfo(){
    System.out.println(id+" "+name);
    System.out.println("answers are:");
    Iterator<String> itr=answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}

}
```

Beans.xml

The list element of constructor-arg is used here to define the list.

```
<?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="q" class="Question">
```

```
<property name="id" value="1"></property>
<property name="name" value="What is Java?"></property>
<property name="answers">
<list>
<value>Java is a programming language</value>
<value>Java is a platform</value>
<value>Java is an Island</value>
</list>
</property>
</bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
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;

public class Test {
public static void main(String[] args) {
    Resource r=new ClassPathResource("Beans.xml");
    BeanFactory factory=new XmlBeanFactory(r);

    Question q=(Question)factory.getBean("q");
    q.displayInfo();

}
}
```

Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Advantage of Autowiring

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

No control of programmer.

Autowiring Modes

There are many autowiring modes:

No.	Mode	Description
1)	no	It is the default autowiring mode. It means no autowiring bydefault.
2)	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3)	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4)	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
5)	autodetect	It is deprecated since Spring 3.

Step 1: Create a Maven Project

group id : com.spring.autowiring

Artifact id: Spring-AutoWiring

Step 2: Open pom.xml and add the dependency of spring-context

Step 3: Create a package com.sping.autowiring

Step 4: Inside the package - Create Interface Pizza, VegPizza.java, NonVegPizza.java, Test.java

Step 5: Create the Beans.xml

Example of Autowiring

Pizza interface

```
package com.spring.autowiring;

public interface Pizza {
	String getPizza();
}
```

VegPizza.java

```
package com.spring.autowiring;


public class VegPizza implements Pizza{

	public String getPizza() {
		// TODO Auto-generated method stub
		return "Veg Pizza";
	}


}
```

NonVegPizza.java

```
package com.spring.autowiring;

public class NonVegPizza implements Pizza{

	public String getPizza() {
		// TODO Auto-generated method stub
		return "Non Veg Pizza";
	}


}
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.spring.autowiring</groupId>
  <artifactId>Spring-AutoWiring</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>

<groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.0.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Beans.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="b"
class="com.spring.autowiring.VegPizza"></bean>
<bean id="a"
class="com.spring.autowiring.NonVegPizza"
autowire="byName"></bean>
</beans>
```

Test.java

```
package com.spring.autowiring;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlAp
plicationContext;

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
        NonVegPizza obj =
(NonVegPizza)context.getBean("a");
        System.out.println(obj.getPizza());
```

```
    }

}
```

1) byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

```
1. <bean id="b" class="org.sssit.B"></bean>
2. <bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

```
1. <bean id="b1" class="org.sssit.B"></bean>
2. <bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

```
1. <bean id="b1" class="org.sssit.B"></bean>
```

```
2. <bean id="a" class="org.sssit.A" autowire="byType"></
   bean>
```

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

```
1. <bean id="b1" class="org.sssit.B"></bean>
2. <bean id="b2" class="org.sssit.B"></bean>
3. <bean id="a" class="org.sssit.A" autowire="byName"></
   bean>
```

In such case, it will throw exception.

3) constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
1. <bean id="b" class="org.sssit.B"></bean>
2. <bean id="a" class="org.sssit.A" autowire="constructor"></
   bean>
```

4) no autowiring mode

In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

1. <bean id="b" class="org.sssit.B"></bean>
2. <bean id="a" class="org.sssit.A" autowire="no"></bean>

Inner Bean Example

This is an example of how to use an inner Bean definition inside a Bean. In Spring 3.2.3, when a bean is only used as a property of another bean it can be declared as an inner bean. Spring's XML-based configuration metadata provides the use of `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements of a bean definition, in order to define the so-called inner bean.

```
<project xmlns="http://maven.apache.org/POM/4.0.0";
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/
4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.spring.InnerBean</groupId>
    <artifactId>springInnerBean</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.1.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.0.RELEASE</version>
        </dependency>
    </dependencies>

  </project>
```

Create a simple Spring Bean with an inner bean

We create a simple Spring Bean, that is `HelloWorld`, and has one property, that is another bean, `Foo`.
HelloWorld.java:

```
package com.spring.InnerBean;
public class HelloWorld {

    private Foo foo;

    public void setFoo(Foo foo) {
        this.foo = foo;
    }

    public String toString(){
        return "HelloWorld! \n" + foo;
    }

}
```

The inner bean `Foo` is the one below:
Foo.java:

```
package com.spring.InnerBean;

public class Foo {

private String name;

public String getName() {
    return name;
}
```

```
public void setName(String name) {
    this.name = name;
}

@Override
public String toString(){
    return "I am " + name + " of inner bean foo.";
}
}
```

XML-based approach for inner bean

Usually, when a bean is used by another bean, the common way to declare it inside the other bean definition by using the `<ref>` attribute, as mentioned in a previous example. But when the bean is only used as a property to another bean, then it can be used as an inner bean and is only declared as a property of the other bean. Here, `fooBean` is only used by `helloWorldBean`, so it is an inner bean of `helloWorldBean`.
The inner bean is supported both in setter injection `<property/>` and constructor injection `<constructor-arg>` elements.

4.1 Using setter injection <property/> element

With setter injection in `helloWorldBean`, `fooBean` is defined in the `<property/>` element, as shown below:
HelloWorld.java:

```
    private Foo foo;

    public void setFoo(Foo foo) {
        this.foo = foo;
    }

    public String toString(){
        return "HelloWorld! \n" + foo;
    }
```

```
}
```

applicationContext.xml

```
<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"
    xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:task="http://www.springframework.org/schema/task" xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-3.2.xsd">

<bean id="helloWorldBean" class="package com.spring.InnerBean.HelloWorld">
    <property name="foo">
```

```
            <bean class="package
com.spring.InnerBean.Foo">
                <property name="name"
value="fooName"/>
            </bean>
        </property>
    </bean>

    </beans>
```

Using constructor injection <constructor-arg>

With constructor injection, the inner bean is defined in the `<constructor-arg>` element, as shown below:
HelloWorld.java:

```
public class HelloWorld {

    private Foo foo;

    public HelloWorld(Foo foo) {
        this.foo = foo;
    }

    public void setFoo(Foo foo) {
        this.foo = foo;
    }

    public String toString(){
        return "HelloWorld! \n" + foo;
    }

}
```

applicationContext.xml

```
<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"
    xmlns:aop="http://www.springframework.org/schema/
aop" xmlns:context="http://www.springframework.org/
schema/context"
    xmlns:jee="http://www.springframework.org/schema/
jee" xmlns:tx="http://www.springframework.org/schema/
tx"
    xmlns:task="http://www.springframework.org/
schema/task" xsi:schemaLocation="http://
www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-3.2.xsd
http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-
beans-3.2.xsd http://www.springframework.org/schema/
context http://www.springframework.org/schema/
context/spring-context-3.2.xsd http://
www.springframework.org/schema/jee http://
www.springframework.org/schema/jee/spring-jee-3.2.xsd
http://www.springframework.org/schema/tx http://
www.springframework.org/schema/tx/spring-tx-3.2.xsd
http://www.springframework.org/schema/task http://
www.springframework.org/schema/task/spring-
task-3.2.xsd">

    <bean id="helloWorldBean" class="package
com.spring.InnerBean.HelloWorld">
        <constructor-arg>
            <bean class="package
com.spring.InnerBean.Foo">
                <property name="name" value="fooName"
/>
            </bean>
```

```
        </constructor-arg>
    </bean>
</beans>
```

Run the application

Through the `ApplicationContext` the beans are loaded to `App.class`.
App.java:

```
import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlAppli
cationContext;

import com.spring.InnerBean.HelloWorld;

public class App {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

            ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xm
l");
            HelloWorld helloWorld = (HelloWorld)
context.getBean("helloWorldBean");
            System.out.println(helloWorld);
    }
}
```

Output:

```
HelloWorld!
I am fooName of inner bean foo.
```

Spring Bean Definition Inheritance & Bean Definition Template

- Create your project with name SpringEx and a package com.example. This should be under the src folder of your created project.
- Add the Spring Libraries that are required using the Add External JARs options.
- Create HelloWorld.java, HelloIndia.java and MainApp.java under the above made package.
- Write the Beans.xml configuration file under the src folder.
- Finally write code for all **Java files** and Bean config file and run the application as described.

HelloWorld.java

```
package com.example;
public class HelloWorld {
   private String message1;
   private String message2;
   public void setMessage1(String message){
      this.message1 = message;
   }
   public void setMessage2(String message){
      this.message2 = message;
   }
   public void getMessage1(){
      System.out.println("World Message1 : " + message1);
   }
   public void getMessage2(){
      System.out.println("World Message2 : " + message2);
   }
}
```

HelloIndia.java

```
package com.example;
public class HelloIndia {
   private String message1;
   private String message2;
   private String message3;
   public void setMessage1(String message){
      this.message1 = message;
   }
   public void setMessage2(String message){
      this.message2 = message;
   }
   public void setMessage3(String message){
      this.message3 = message;
   }
   public void getMessage1(){
      System.out.println("India Message1 : " + message1);
   }
   public void getMessage2(){
      System.out.println("India Message2 : " + message2);
   }
   public void getMessage3(){
      System.out.println("India Message3 : " + message3);
   }
}
```

MainApp.java

```
package com.example;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCon
text;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld objA = (HelloWorld)
context.getBean("helloWorld");
      objA.getMessage1();
```

```
    objA.getMessage2();
    HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
    objB.getMessage1();
    objB.getMessage2();
    objB.getMessage3();
  }
}
```

Output:

```
World Message1 : Hello World!
World Message2 : Hello Second World!
India Message1 : Hello India!
India Message2 : Hello Second World!
India Message3 : Namaste India!
```

BeanTemplate – Bean Inheritance

```
<?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 = "beanTeamplate" abstract = "true">
    <property name = "message1" value = "Hello World!"/>
    <property name = "message2" value = "Hello Second World!"/>
    <property name = "message3" value = "Namaste India!"/>
  </bean>
    <bean id = "helloIndia" class = "com.example.HelloIndia"
parent = "beanTeamplate">
    <property name = "message1" value = "Hello India!"/>
    <property name = "message3" value = "Namaste India!"/>
  </bean>
</beans>
```