DURGA SOFTWARE SOLUTIONS

# Practical approach to Spring Framework

## Spring Framework 3.0

**Mr. Sriman**

As part of this we cover Spring Core, AOP, Spring JDBC, Transactions, Spring MVC and ORM. This includes all the versions of spring starting with 2.0, 2.5, 3.0 and latest 3.1.1 (Spring Annotations as well).

# Contents

# XML

# 1 XML

XML stands for extensible markup language. Markup language is the language using which you can build other languages like, HTML, XML.

XML is defined and governed by W3Org. The first and final version of XML is XML 1.0. XML is the document which represents data. Unlike C, C++, Java etc. XML is not a programing language. It is the defacto standard for carrying information between computer systems.

Every language has keywords. If you take example as C, it has keywords like (if, for, while, do, break, continue etc.) but when it comes to XML there are no keywords or reserved words. What you write will become the element of XML language.

## 1.1 XML Element

Element in an XML is written in angular braces for e.g. <beans>. In XML there are two types of elements as follows

1) Start Element/Opening Tag: - Start element is the element which is written in <elementname> indicating the start of a block.
2) End Element/End Tag: - End element is the element which is written in </elementname> indicating the end of a block.

As everything is written in terms of start and end elements, XML is said to be more structured in nature. An XML Element may contain content or may contain other elements under it. So, XML elements which contain other elements in it are called as Compound elements or XML Containers.

## 1.2 XML Attribute

If we want to have supplementary information attach to an element, instead of having it as content or another element, we can write it as an Attribute of the element.

*Example:-*

```
<bean name="pilot" class="Pilot">
  <constructor-arg ref="plane"/>
</bean>
```

In the above example "bean" is an element which contains two attributes name and class which acts as an supplementary information. Constructor-arg is a sub-element under "bean" element.

## 1.3 Well-formness

As how any programing language has syntax in writing its code, Well-formness of XML document talks about how to write an XML document. Well-formness indicates the readability nature of an XML document. In other way if an XML document is said to be well-formed then it is readable in nature.

Following are the rules that describe the Well-formness of an XML Document.

1) Every XML document must start with **PROLOG**: - prolog stands for processing instruction, and typically used for understanding about the version of XML used and the data encoding used.

   *Example*: - <?xml version="1.0" encoding="utf-8"?>

2) Root Element: - XML document must contain a root element, and should be the only one root element. All the other elements should be the children of root element.
3) Level constraint: - Every start element must have an end element and the level at which you open a start element, the same level you need to close your end element as well.

*Example:-*

```
<?xml version="1.0" encoding="utf-8"?>
<student>
        <info>
                <rollno>42</rollno>
                <name>John</info>  (-- info is closed in-correctly --)
        </name>
<student>
```

If any XML is said to follow the above defined rules then it is termed as well-formed.

## 1.4 XML Usage

An XML document is used in two scenarios

1) **Used for transferring information**:- As said earlier XML is a document representing data and is used for carrying information between two computer systems in an Interoperable manner.
2) **Configurations**: - In J2EE world every component/resource you develop like Servlet or EJB, it has to be deployed into a Web Server. For e.g. in a Servlet application, the path with which the servlet has to be accessible should be specified to the Servlet container so that it can map the incoming request to the Servlet. This is done in a web.xml. When we provide the configuration

information in a XML file the main advantage is, the same xml document can be used in different platforms without changing anything.

## 1.5 Validatity

Every XML in-order to parse should be well-formed in nature. As said earlier well-formness of an XML document indicates whether it is readable or not, it doesn't talks about wheather the data contained in it is valid or not.

Validity of the XML document would be defined by the application which is going to process your XML document. Let's consider a scenario as follows.

You want to get driving license. In order to get a driving license you need to follow certain process like filling the RTA forms and signing them and submitting to the RTA department.

Instead of this can you write your own format of letter requesting the driving license from an RTA department, which seems to be not relevant because driving license is something that would be issued by the RTA department. So, the whole and sole authority of defining what should a person has to provide data to get a driving license will lies in the hands of RTA department rather than you.

In the same way when an application is going to process your xml document, the authority of defining what should be there as part of that xml is lies in the hands of the application which is going to process your document.

For example let's consider the below xml fragment.

*Example:- PurchaseOrder xml document*

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
  <orderItems>
        <item>
              <itemCode>IC323</itemCode>
              <quantity>24</quantity>
        </item>
        <item>
              <itemCode>IC324</itemCode>
              <quantity>abc</quantity>
        </item>
  </orderItems>
<purchaseOrder>
```

In the above xml even though it confirmes to all the well-formness rules, it cannot be used for business transaction, as the 2nd <quantity> element carries the data as "abc" which doesn't makes any sense.

So, in order to check for data validity we need to define the validation criteria of an XML document in either a DTD or XSD document.

## 2 DTD

DTD stands for Document Type Definition. It is the document which defines the structure of an XML document.

As we have two types of XML elements Simple and Compound elements we need to represent in DTD, my xml contains these elements which are of either simple or compound.

So, we need to understand the syntax of how to declare a simple element of an xml in dtd as well as compound element.

Syntax for Simple Element of an XML

<!Element elementname (#PCDATA)>

Here we declared that my xml should contain an element whose name is elementname which contains parse able character data.

Syntax for Compound elemtent of an XML

<!Element elementname (sub-elem1, sub-elem2…)

Here we declared I have an element "elementname" which contains sub elements under it sub-elem1, sub-elem2 etc.

For example the sample DTD document for an xml is shown below.

XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
 <orderItems>
        <item>
                <itemCode>IC323</itemCode>
                <quantity>24</quantity>
        </item>
        <item>
                <itemCode>IC324</itemCode>
                <quantity>abc</quantity>
        </item>
 </orderItems>
<purchaseOrder>
```

For the above xml the DTD looks as shown below.

```
<?xml version=”1.0” encoding=”utf-8”?>
<!Element purchaseOrder (orderItems)>
<!Element orderItems (item+)>
<!Element item (itemCode, quantity)>
<!Element itemCode (#PCDATA)>
<!Element quantity (#PCDATA)>
```

In the above xml if you observe the orderItems element can contain any number of item elements in it, but atleast one item element must be there for an purchaseOrder. This is called occurance of an element under another element, to indicate this we use three symbols.

? – Represents the sub element under a parent element can appear zero or one-time (0/1).

+ - indicates the sub element must appear at least once and can repeat any number of times (1 – N)

* - indicates the sub element is optional and can repeat any number of times (0 - N)

You will mark the occurrence of an element under another element as follows.

<!Element elementname (sub-elem1 (?/+/*), sub-elem2(?/+/*)>

Leaving any element without any symbol indicates it is mandatory and at max can repeat only once.

## 2.1 Drawback with DTD’s.

DTD are not typesafe, which means when we declare simple elements we indicate it should contain data of type (#PCDATA). #PCDATA means parsable character data means any data that is computer represented format. So it indicates an element can contain any type of data irrespective of whether it is int or float or string. You cannot impose stating my element should contain int type data or float. This is the limitation with DTD documents.

# 3 XML Schema Document (XSD)

XSD stands for XML schema document. XSD is also an XML document. It is owned by W3Org. The latest version of XSD is 1.1.

Even though we can use DTD's for defining the validity of an XML document, it is not type safe and is not flexibile. XSD is more powerful and is type strict in nature.

XSD document is used for defining the structure of an XML document; it declares elements and the type of data that elements carry.

As XSD is also an XML document so, it also starts with prolog and has one and only one root element. In case of XSD document the root element is <schema>. All the subsequent elements will be declared under this root element.

An XML contains two types of elements. A) Simple elements (these contains content as data) B) Compound or Container's (these contains sub-elements under it).

So, while writing an XSD documents we need to represent two types of elements simple or compound elements. The syntax's for representing a simple and compound elements of XML in a XSD document are shown below.

Syntax for representing simple elements of XML document

<xs:element name="elementname" type="datatype"/>

Syntax for representing compound element of XML document

In order to represent compound element of an XML document in an XSD, first we need to create a type declaration representing structure of that xml element. Then we need to declare element of that user-defined type, shown below.

```
<xs:complexType name="typeName">
   <xs:sequence> or <xs:all>
         <xs:element name=".." type=".."/>
         <xs:element name=".." type=".."/>
         <xs:element name=".." type=".."/>
   </xs:sequence> or </xs:all>
</xs:complexType>
```

Declaring a complex type is equal to declaring a class in java. In java we define user-defined data types using class declaration. When we declare a class it represents the structure of our data type, but it doesn't allocates any memory. Once a class has been declared you can create any number of objects of that class.

The same can be applied in case of complex type declaration in an XSD document. In order to create user-defined data type in XSD document you need to declare a

complex type, creating a complex indicates you just declared the class structure. Once you create your own data type, now you can define as many elements you need of that type. Let's say for example I declared a complex type whose name is AddressType as follows

```
<xs:complexType name="AddressType">
   <xs:sequence>
         <xs:element name="addressLine1" type="xs:string"/>
         <xs:element name="addressLine2" type="xs:string"/>
         <xs:element name="city" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Now the above fragment indicates you created your own user-defined data type whose name is "AddressType". Now you can declare any number of elements representing that type as shown below.

```
<xs:element name="myAddress" type="AddressType"/>
```

Let's take an XML document for which we will show how its XSD looks like

XML Document – Courier consignment information

```
<consignment>
   <id>C4242</id>
   <bookedby>durga</bookedby>
   <deliveredTo>Sriman</deliveredTo>
   <shippingAddress>
         <addressLine1>S.R Nagar</addressLine1>
         <addressLine2>Opp Chaitanya</addressLine2>
         <city>Hyderabad</city>
         <state>Andhra Pradesh</state>
         <zip>353</zip>
         <country>India</country>
   </shippingAddress>
</consignment>
```

XSD Document – Courier consignment information

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="consignment" type="consignmentType"/>
   <xs:complexType name="consignmentType">
         <xs:sequence>
               <xs:element name="id" type="xs:string"/>
               <xs:element name="bookedby" type="xs:string"/>
               <xs:element name="deliveredTo" type="xs:string"/>
               <xs:element name="shippingAddress"
type="shippingAddressType"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="shippingAddressType">
         <xs:sequence>
               <xs:element name="addressLine1" type="xs:string">
               <xs:element name="addressLine2" type="xs:string">
               <xs:element name="city" type="xs:string">
               <xs:element name="state" type="xs:string">
               <xs:element name="zip" type="xs:int">
               <xs:element name="country" type="xs:string">
         </xs:sequence>
   </xs:complexType>
</xs:schema>
```

## 3.1 Sequence VS All

When declaring a complex type, following the <xs:complexType> tag we see a tag <xs:sequence> or <xs:all>. When we use <xs:sequence> under <xs:complexType> tag, what it indicates is all the elements that are declared in that complex type must appear in the same order in xml document. For example let's take a complex type declaration "ItemType" which uses <xs:sequence> tag in its declaration.

```
<xs:element name="item" type="ItemType"/>
<xs:complexType name="ItemType">
   <xs:sequence>
         <xs:element name="itemCode" type="xs:string"/>
         <xs:element name="quantity" type="xs:int"/>
   </xs:sequence>
</xs:complexType>
```

So while using the item element in the xml we need to declare the itemCode and quantity sub-elements under item in the same order as shown below.

```
<item>
   <itemCode>IC303</itemCode>
   <quantity>35</quantity>
</item>
```

When we use <xs:all>, under item element the itemCode and quantity may not appear in the same order. First quantity might appear then itemCode can present in the xml.

# 4 XSD Namespace

Every programming language one or in another way allows you to declare user-defined data types. Let's consider the case of "C" language it allows you to declare your own types using Structure. In case of "C++" Class declaration allows you to declare your own type, same in case of Java as well.

When it comes to XSD you can declare your element type using XSD complex type declaration.

As how any language allows you to create your own types, they allow you to resolve the naming collision between their types. Let's consider the case of Java; it allows you to declare your own types by class declarations. But when a programmer is given a choice of declaring their own types, language has to provide a means to resolve type names collision declared by several programmers. Java allows resolving those type naming conflicts using packages.

Packages are the means of grouping together the related types. It will allow you to uniquely identify a type by prefixing the package name. In the same way XSD also allows you to declare your own data type by using Complex Type declaration and in the same way it allows you to resolve the type naming conflicts by means of Namespace declaration.

XSD Namespaces has two faces,

1) Declaring the namespace in the XSD document using Targetnamespace declaration

2) Using the elements that are declared under a namespace in xml document.

## 4.1 XSD Targetnamespace

Targetnamespace declaration is similar to a package declaration in java. You will bind your classes to a package so, that while using them you will refer with fully qualified name. Similarly when you create a complexType or an Element you will bind them to a namespace, so that while referring them you need to use qName.

In order to declare a package we use package keyword followed by name of the package. To declare a namespace in XSD we need to use targetNameSpace attribute at the Schema level followed by targetnamespace label as shown below.

*Example:-*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace="http://durgasoft.com/training/calendar/types">
  <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
  </xs:complexType>
</xs:schema>
```

The courseType by default is binded to the namespace http://durgasoft.com/training/calendar/types

So, while creating an element "course" of type courseType you should use the qName of the courseType rather simple name. (qName means namespace:element/type name).

But the namespace labels could be any of the characters in length, so if we prefix the entire namespace label to element/type names it would become tough to read. So, to avoid this problem XSD has introduced a concept called short name. Instead of referring to namespace labels you can define a short name for that namespace label using xmlns declaration at the <schema> level and you can use the short name as prefix instead of the complete namespace label as shown below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace=http://durgasoft.com/training/calendar/types
xmlns:dt="http://durgasoft.com/training/calendar/types">
  <xs:element name="course" type="dt:courseType"/>
  <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
  </xs:complexType>
</xs:schema>
```

In java we can have only one package declaration at a class level. In the same way we can have only one targetNamespace declaration at an XSD document.

## 4.2 Using elements from an xml namespace (xmlns)

While writing an XML document, you need to link it to XSD to indicate it is following the structure defined in that XSD. In order to link an XML to an XSD we use an attribute "schemaLocation".

schemaLocation attribute declaration has two pieces of information. First part is representing he namespace from which you are using the elements. Second is the document which contains this namespace, shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

In the above xml we stated this xml is using xsd document courseInfo.xsd whose namespace is http://durgasoft.com/training/calendar/types

If you want to include two XSD documents in the xml then you need to declare in schemaLocation tag <namespace1> <schemalocation1> <namespace2> <schemalocation2>.
*For example:-*

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

Now in the above xml we are using two XSD documents courseInfo.xsd and vacation.xsd. With this declaration we will not be able to find whether the course element is used from courseInfo.xsd or vacation.xsd. To indicate it we should prefix the namespace to the course element.

But the namespace labels can be arbitrary string of characters in any length. So, we need to define short name, so that we can prefix shortname while referring the elements as shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<dc:course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance
xmlns:dc="http://durgasoft.com/training/calendar/types">

</dc:course>
```

## 4.3 Difference between DTD and XSD

| DTD | XSD |
|---|---|
| • DTD stands for document type definition<br>• DTD's are not XML Type documents<br>• DTD's are tough to learn as those are not XML documents. So, an XML programmer has to under new syntaxes in coding DTD<br>• DTD's are not type safe, these will represents all the elements with data type as (#PCDATA).<br>• DTD's don't allow you to create use-defined types. | • XSD stands for XML Schema Documents<br>• XSD's are XML Type documents<br>• As XSD's are XML type documents it is easy for any programmer to work with XSD.<br>• XSD's are strictly typed, where the language has list of pre-defined data types in it. While declaring the element you need tell whether this element is of what type.<br>• XSD's allows you to create user-defined data types using complexType declaration and using that type you can create any number of elements. |

# Spring Framework

# 5 Spring Framework

## 5.1 Introduction to Spring Framework

Spring Framework is an open source, light weight, loosely coupled Java Framework that allows you to build enterprise applications. Spring has been started in year 2003, it is not a one kind of framework which supports development of certain areas of J2EE, and it is a complete framework.

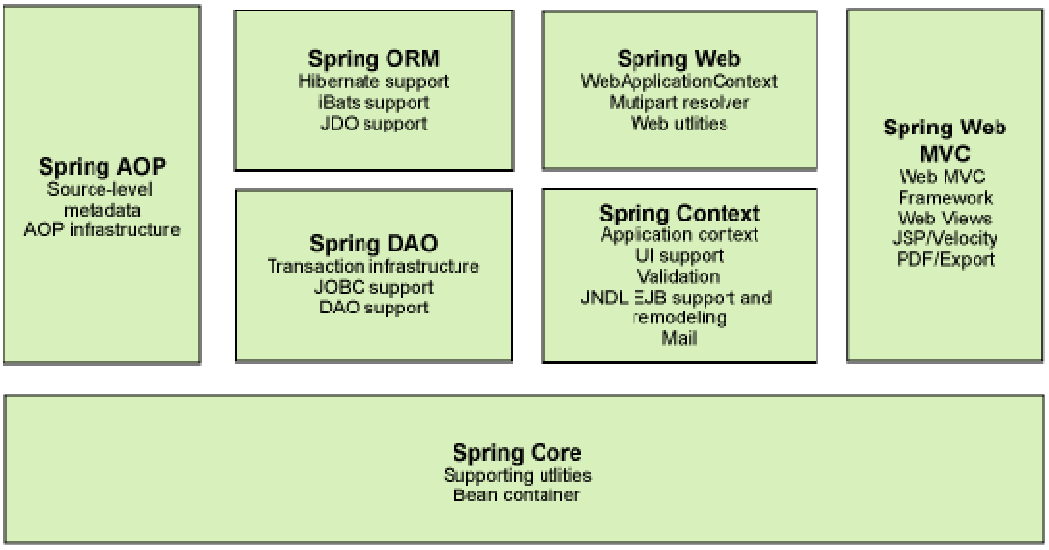
Using spring you can develop various types of application (in par with J2EE) like core java, remoting, web applications etc. So, you can think spring as a layer above your J2EE, but never spring is a replacement for the J2EE, it always compliments your J2EE in facilitation in quicker development.

Spring handles the infrastructure so you can focus on your application. Spring enables you to build applications from POJO (Plan old java objects), by providing enterprise services (like logging, transaction etc) non-invasively. This indicates your application business logic is free from spring specific classes or interfaces, at any point of time you can easily move out of spring without making any code changes.

Few of the examples of spring platform advantages

1) Pojo methods can work within database transaction without dealing with Transaction API's
2) Pojo methods can be exposed as remote methods without dealing with RMI API's
3) Pojo methods can behave like a message handler without dealing with JMS API's

Main advantage of spring is its layered architecture. Even its supports various types of application development, it has been designed keeping in mind to be light weight by dividing it into various modules as shown below.



Based on your application needs you can select and pick the module of your choice for development, even you can use combination of multiple modules. Following are the functionalities of each module.

1. Spring Core: - The core module is the heart of spring framework, it spans horizontally across all the other modules. Core module provides the essential functionality for spring framework. The primary component of core module is BeanFactory, an implementation of Factory pattern. BeanFactory allows you to separate your configuration information and dependency specification from your actual application code. Using spring core you can develop a core java application as well.
2. Spring Context: - The spring context module is a configuration file that allows you to provide the context information to the framework. The spring context may contain configuration about the pojo's, jndi, email, internationalization etc.
3. Spring AOP: - AOP stands for aspect oriented programming; it's a new programming technic that allows programmers to induce cross-cutting logic across several components of your application. Cross-cutting logic implies any logic or code that has to be applied across several components. For example transactions, logging, auditing and security are some of the examples of applications of AOP.
4. Spring DAO: - It is an abstraction of JDBC DAO. In an traditional JDBC application you have to write lot of boiler plate code like getting the connection, executing the statements, iterating over the result sets and managing the resources (e.g.. connection, statement, resultset etc.). Along with this while working with JDBC code you need to handle lot of annoying checked exceptions, spring JDBC avoids lot of boiler plate code and it can manages resources as well as has a meaningful exception hierarchy defined in the framework to handle several types of exceptions that JDBC code might throw.
5. Spring ORM: - spring framework plugins to several ORM frameworks to provide its Object Relation tool, the idea behind spring framework is it don't want to re-invent the wheel rather wants to make existing tools to be used easily in their applications. So, as part of this effort it has provided integrations to lot of ORM frameworks like Hibernate, iBatis, JPA etc.
6. Spring Web module:- The web context module is built on top application context module, providing the context for web-based applications. This module allows spring to integrate with various other web based frameworks like jakarta struts etc.
7. Spring Web MVC framework: - This module allows us to build Model-View-Controller architecture based applications which contains full features for building Web applications. It varies in many ways from normal web application frameworks like struts etc. The main advantage of going with spring web mvc module is view technology is abstracted from the client and you can have anything as a presentation tier.

By the above we understood that spring core is the foundation module on top of which other modules has been designed, so it is important to understand spring core. Considering the importance of spring core module we have divided it into two parts I) Spring Core Basic and ii) Spring Core Advanced module.
With this separation spring core basic module acts as a foundation, which enables us to work on other modules (without spring core advanced). Advanced module covers the powerful features of spring core and is used rarely when the application demands.

# 6 Spring Core (Basic)

Spring core basic part is foundation which contains essential features of spring framework. The heart of spring framework is IOC (Inversion of Control), which manages dependencies between objects and their lifecycle. In order to benefit out of IOC spring recommends every application to follow certain design guidelines or principles (strategy pattern) to make their applications loosely coupled.

So, let's first examine the design principles and then we focus on Spring Core IOC.

## 6.1 Strategy pattern (Design principle spring recommends)

Strategy pattern lets you build software as loosely coupled collection of interchangeable parts, in contrast with tightly coupled system. This loosely coupling makes your software much more flexible, extensible, maintainable and reusable.

Strategy pattern recommends mainly three principles every application should follow to get the above benefits those are as follows

- Favor Composition over Inheritance
- Code should be open for extension and closed for modification
- Always design to Interfaces never code to Implementation

Let us examine all the three principles by taking an example

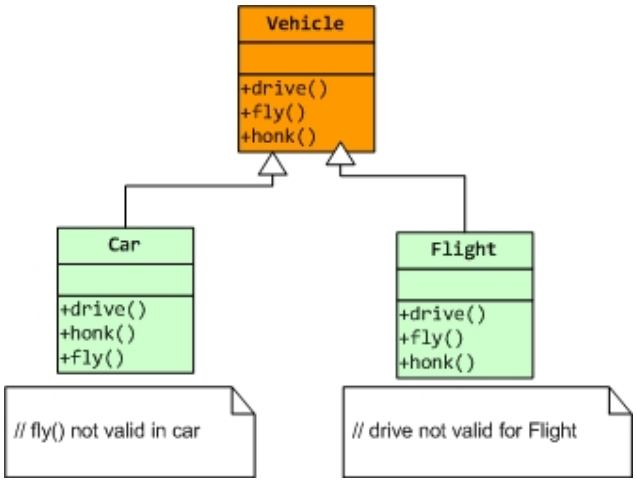
A core java application contains multiple classes. Each class contains logic for performing some kind of task. A pojo class which contains logic for performing operation is called pojo based business logic class.

Every class in order to perform a task has to talk with other class to get the things done. A class can use the functionality of other class in two ways.

1) Through Inheritance – Inheritance is the process of extending once class from another class to get its functionalities.
2) Composition – A class will hold reference of other classes as attributes inside it.

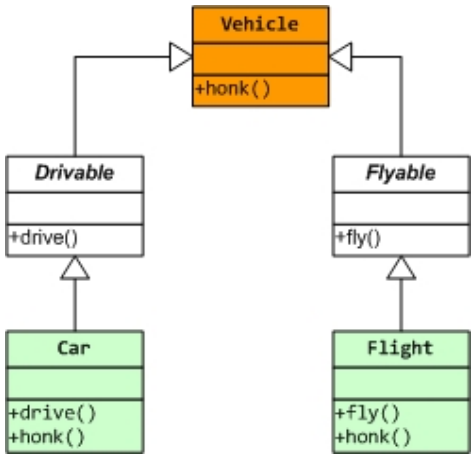
Using Inheritance you always express relation between two classes as IS-A relationship. But always not every relationship can be expressed in IS-A relationship.

The problem with most of the programmers is they will always choose the option of inheritance to use the functionality of other classes. But this may not be apt in all cases; we have to go for inheritance only when all the behaviors (methods) of your base class can be shared across derived classes. Refer to the below example explaining the same.



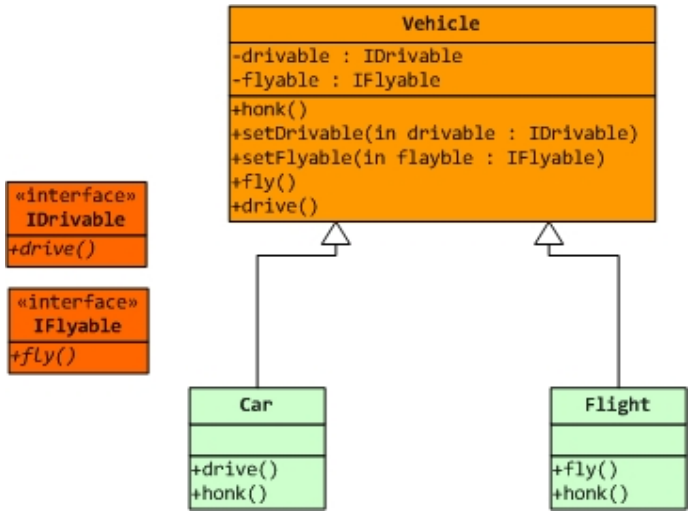

In the above diagram I have a vehicle which contain drive (), fly () and blow horn (honk ()). So I want to represent a Car and a Flight which is of Vehicle type. If I try to express this with Inheritance I will end up in implementing fly method in Car class as empty and drive method in flight class as empty (because the fly and drive are in-appropriate for Car and Flight respectively).

So to overcome this modifies your class structure into two hierarchies as Drivable and Flyable making Vehicle as sub-classes shown below.




But the problem with above approach is your design model is rigid (closed). Through the above design you are stating you can have a vehicle which can either fly or drive. But in future if I have a vehicle which can drive as well as fly then it is not possible to represent that kind of vehicle with the above design. This may leads to re-designing the entire application.

So, by the above example it is clear that we can't represent all the problems through inheritance. In order to solve this Composition would be more apt than Inheritance. Below diagram shows the solution for the above described problem.




In the above design, Vehicle is an abstract class which contains IDrivable and IFlyable as attributes. Along with this vehicle contains honk () as implemented method as it shares across all its derived classes.
Now Vehicle declares two protected methods drive and fly which in-turn calls the methods on drivable and flyable attributes.

Now when we create a Car, Car will override the drive method where it will sets the implementation Object IDrivable class (for e.g. DefaultDrivable is implementing from IDrivable) to the drivable attribute of super class through its setter and then makes a call to super.drive().

The same applies to Flight as well.

With the above design we understood the above problem can be solved using composition rather than Inheritance. Your design is so flexible that instead of having your vehicle of type only Drivable or Flyable it is now more open to create both the types. As the IDrivable and IFlyable are interfaces, you can switch between various implementations of those interfaces. So, your vehicle is not talking to particular Drivable or Flyable rather it is talking to interfaces.

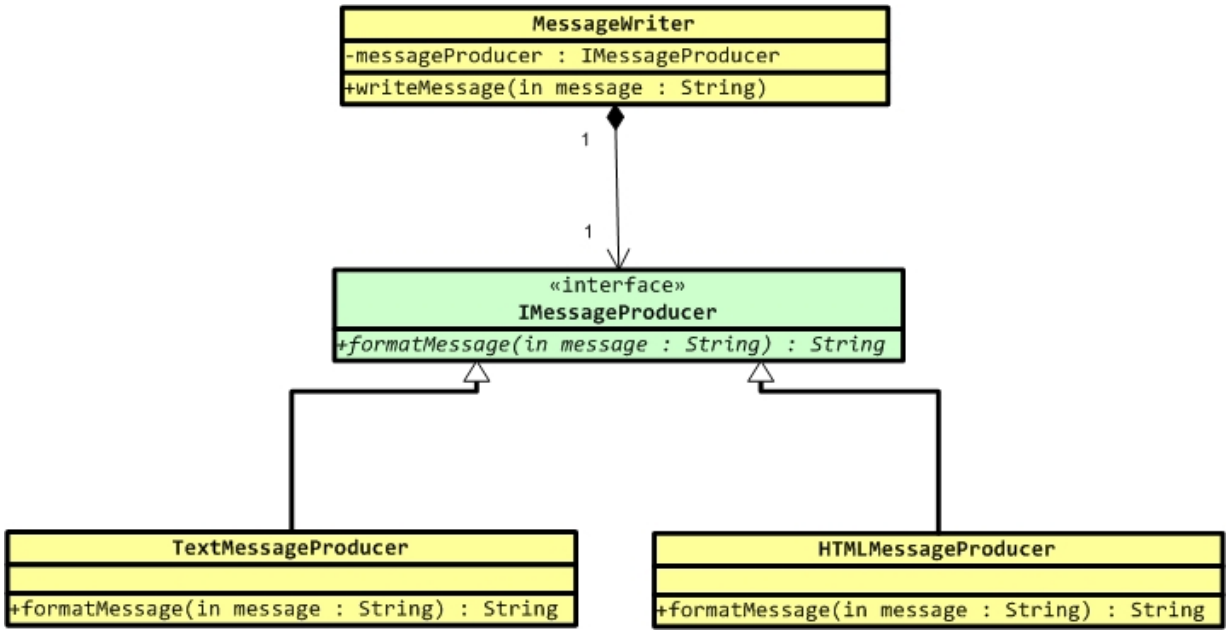
## 6.2 Spring Inversion of Control (IOC)

IOC stands for Inversion of control, which is a principle also known as Dependency Injection. It is the process where every object defines their dependency with whom they work with in a configuration file. These dependent objects are injected through Constructor arguments or method parameters or factory methods. The container injects the declared dependencies while creating the beans. As the process is inverse where the bean is not acquiring the dependency rather the container is pushing the dependencies hence it is called inversion of control.

Let's understand the above principle by taking an example.

We have a MessageWriter class which will write the message to the console. The MessageWriter class will get the message from MessageProducer class. But we have multiple types of MessageProducers like TextMessageProducer, HTMLMessageProducer etc.

So the MessageWriter will talks to the message producers through an interface IMessageProducer, which defines a method formatMessage. This has been explained in the below diagram.




If you observe carefully the above design has followed all the design principles that spring recommends. Code fragment for the above classes are as shown below.

**IMessageProducer.java**

```
package com.ioc.beans;
public interface IMessageProducer {
        String formatMessage(String message);
}
```

**TextMessageProducer.java**

```
package com.ioc.beans;

public class TextMessageProducer
implements IMessageProducer {
      public String formatMessage(String
message) {
          return "Hello " + message + "!";
      }
}
```

**HTMLMessageProducer.java**

```
package com.ioc.beans;

public class HTMLMessageProducer
implements IMessageProducer {
    public String formatMessage(String
message) {
        return
"<HTML><HEAD></HEAD><BODY>" +
message + "</BODY></HTML>";
    }
}
```

**MessageWriter.java**

```
package com.ioc.beans;

public class MessageWriter {
   private IMessageProducer messageProducer;
   public void writeMessage(String message) {
      // instantiate messageProducer with concrete implementation class
      messageProducer = new HTMLMessageProducer();
      String formattedData = messageProducer.formatMessage(message);
      System.out.println(formattedData);
   }
}
```

**MessageTest.java**

```
package com.ioc.beans;

public class MessageTest {
    public static void main(String args[]) {
        MessageWriter writer = new MessageWriter();
        writer.writeMessage("Welcome to Spring");
    }
}
```

Even you followed the recommended design principles, it has two problems.

- In your MessageWriter class inside the writeMessage method you have hardcoded the concrete class name HTMLMessageProducer while instantiating messageProducer attribute. If you want to switch between HTMLMessageProducer to TextMessageProducer you need to modify the source code of the MessageWriter.
- MessageWriter just wants to use the functionality of IMessageProducer, but still MessageWriter has to know the instantiation process of IMessageProducer implementation class's. This means how to create IMessageProducer implementation class whether to call new operation to create or any factory method should be called to get the object etc.

To avoid the above problems instead of MessageWriter creating the Object of IMessageProducer implementation class, it should externalize this functionality to someone else who is going to create the IMessageProducer objects and injects into MessageWriter. Below code fragment shows the same.

**Modified#1 - MessageWriter.java**

```
package com.ioc.beans;

public class MessageWriter {
  private IMessageProducer messageProducer;
  public void writeMessage(String message) {
     String formattedData = messageProducer.formatMessage(message);
     System.out.println(formattedData);
  }

  public void setMessageProducer(IMessageProducer messageProducer) {
     this.messageProducer = messageProducer;
  }
}
```

**Modified#1 – MessageTest.java**

```
package com.ioc.beans;

public class MessageTest {
   public static void main(String args[]) {
      MessageWriter writer = new MessageWriter();
      IMessageProducer messageProducer = new HTMLMessageProducer();
      writer.setMessageProducer(messageProducer);
      writer.writeMessage("Welcome to Spring");
   }
}
```

If you see the MessageWriter class it is loosely coupled from HTMLMessageProducer and now it can talk to any IMessageProducer implementation classes. Secondly MessageWriter doesn't need to bother about how to instantiate IMessageProducer implementation class, because IMessageProducer implementation class will be injected by calling setter method on it.

Even though we made our business class loosely coupled from specific implementation, but still we hardcoded the HTMLMessageProducer in MessageTest class. If you want to use TextMessageProducer instead of HTMLMessageProducer your main method should be modified to instantiate TextMessageProducer class and should pass it to the MessageWriter via calling its setter.

So, at some place in your code you are hard coding the concrete class references, in order to avoid this you need to use the Spring IOC.

## 6.3 Types of IOC

IOC is of two types; again each type is classified into two more types as follows.

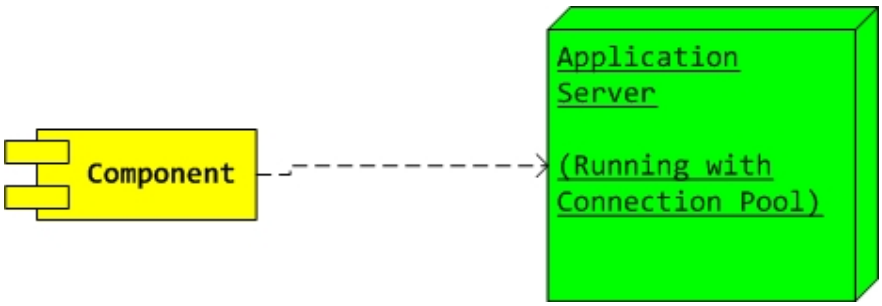
1) Dependency Lookup
    a. Dependency Pull
    b. Contextual Dependency lookup
2) Dependency Injection
    a. Setter Injection
    b. Constructor Injection

### 6.3.1 Dependency Lookup

Dependency lookup is a very old technic which is something exists already and most of the J2EE applications use. In this technic if a class needs another dependent object; it will write the code for getting the dependency. Again this has two variants as said above.

1) Dependency Pull

If we take a J2EE Web application as example, we retrieve and store data from database for displaying the web pages. So, to display data your code requires connection object as dependent, generally the connection object will be retrieved from a Connection Pool. Connection Pools are created and managed on an Application Server. Below figure shows the same.



Your component will look up for the connection from the pool by performing a JNDI lookup. Which means you are writing the code for getting the connection indirectly you are pulling the connection from the pool. As the process is pull, it is called dependency pull.

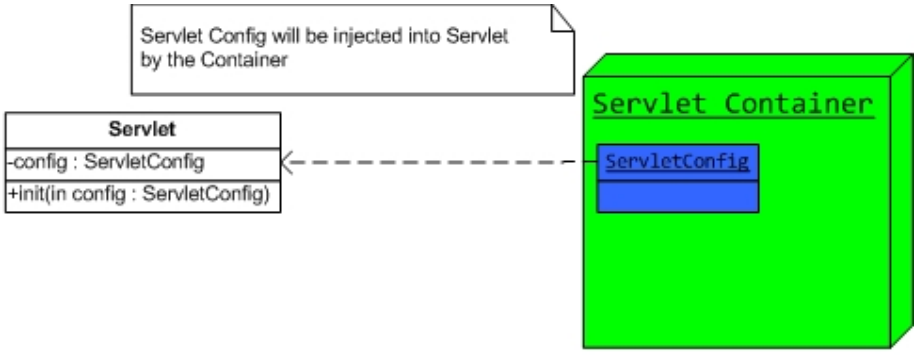
**Pseudo code**

```
Context ctx = new InitialContext();
DataSource ds = ctx.lookup("jndi name of cp");
Connection con = ds.getConnection()
// pulling the connection
```

2) Contextual Dependency Lookup

In this technic your component and your environment/server will agree upon a contract/context, through which the dependent object will be injected into your code.

For e.g If you see a Servlet API, if your servlet has to access environment specific information (init params or to get context) it needs ServletConfig object. But the ServletContainer will create the ServletConfig object. So in order access ServletConfig object in your servlet, you servlet has to implement Servlet interface and override init(ServletConfig) as parameter. Refer the below figure.



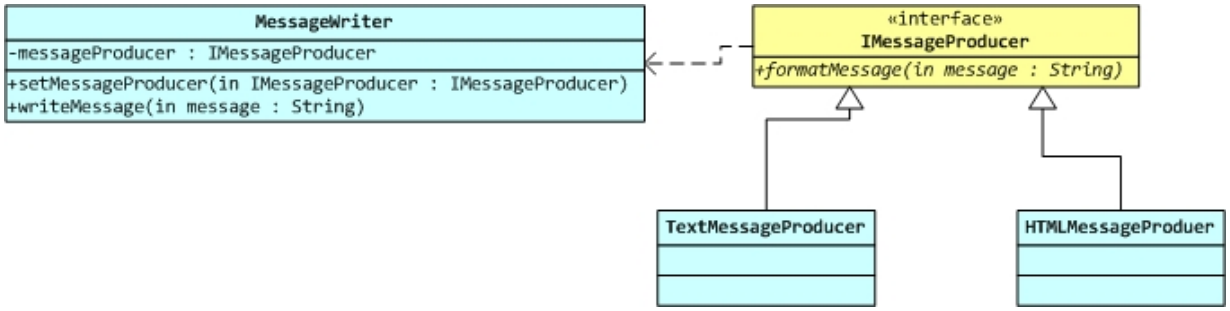

Then the container will pushes the config object by calling the init method of the servlet. Until your code implements from servlet interface, container will not pushes the config object, this indicates servlet interface acts as a contract between you and your servlet so, it is called Context Dependency Lookup (CDL).

### 6.3.2 Dependency Injection

Even though spring supports the above mentioned two technics, the new way of acquiring the dependent objects is using setter injection or constructor injection. This is detailed as below.

1) Setter Injection

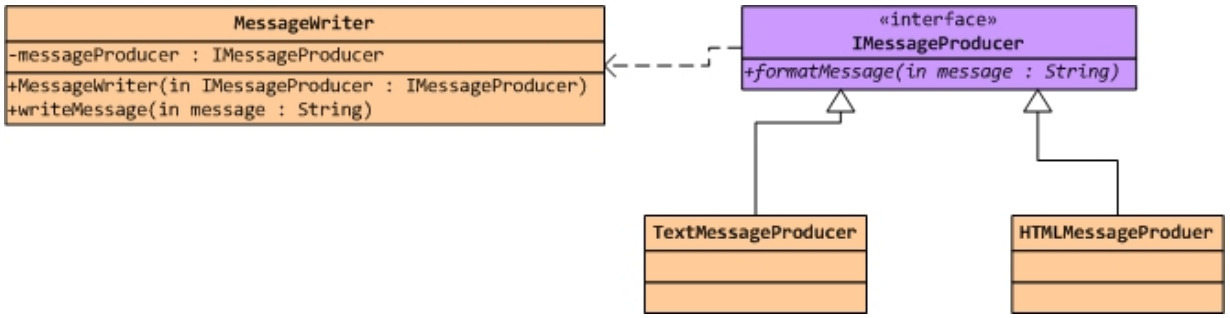
In setter injection if an object depends on another object then the dependent object will be injected into the target class through the setter method that has been exposed on the target classes as shown below.




In the above diagram MessageWriter is the target class onto which the dependent object IMessageProducer implementation will be injected by calling the exposed setter method.

2) Constructor Injection

In this technic instead of exposing a setter, your target class will expose a constructor, which will takes the parameter of your dependent object. As your dependent object gets injected by calling the target class constructor, hence it is called constructor injection as shown below.



Let's modify the earlier example to inject the IMessageProducer into MessageWriter through setter injection and constructor injection.

If we want our class objects to be managed by spring then we should declare our classes as spring beans, any object that is managed by spring is called spring bean. Here the term manages refers to Object creation and Dependency injection (via setter, constructor etc…).

So in our example we want MessageWriter and IMessageProducer implementation classes to be created and injected by spring, so we need to declare them spring in a configuration file called "Spring Beans configuration".

"Spring Bean configuration" is an xml file in which we declare all the classes as beans, so that those will be managed by spring. We need to use <bean> tag to declare a class as spring bean. The below fragment shows how to create a class as spring bean.

**Spring Beans Configuration (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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
       <bean id="textMessageProducer" class="com.ioc.beans.TextMessageProducer"
       />
       <bean id="htmlMessageProducer" class="com.ioc.beans.HTMLMessageProducer"
       />
       <bean id="messageWriter" class="com.ioc.beans.MessageWriter">
       </bean>
</beans>
```

The org.springframework.beans and org.springframework.context packages are the basis for the Spring Framework's IOC Container. The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of Object. So, the above configuration file has to be given to BeanFactory to manage the Objects that are declared in that configuration.

BeanFactory has multiple implementations one of which is XMLBeanFactory which will reads the xml configuration file as resource and creates objects and holds in Memory. This memory representation of objects is called IOC Container. Spring IOC container is an non-physical in-memory object which knows how to instantiate the beans that are declared in the "Spring Bean configuration" and manages their dependencies. The below snippet shows you how to create IOC Container.

**Creating IOC Container**

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                          "com/injection/common/application-context.xml"));
```

Example showing IMessageProducer injected through setter injection into MessageWriter. In our earlier example we have already exposed setter on IMessageProducer class messageProducer attribute. But we are creating both the objects and injecting the messageProducer by explicitly calling the setter in main method.

Instead of us injecting we want spring to create these objects and inject one into another, in order to do this we need to declare all the classes in configuration file including their dependency relations.

Then create IOC Container from which we can get the MessageWriter object and use it. Sample modified #2 version of the code is show below.

**Spring Beans configuration File (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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
        <bean id="textMessageProducer"
                class="com.ioc.beans.TextMessageProducer" />
        <bean id="htmlMessageProducer"
                class="com.ioc.beans.HTMLMessageProducer" />
        <bean id="messageWriter" class="com.ioc.beans.MessageWriter">
                <property name="messageProducer"
                                        ref="htmlMessageProducer"/>
        </bean>
</beans>
```

In order to inject htmlMessageProducer bean into messageWriter bean we need to declare the <property> tag. The name attribute of it refers to the target class dependent attribute. Ref attribute refers to which bean has to be injected into name attribute (by calling its setter's).

**<property>** tag is used for performing setter injection and the dependent object will be injected into target class by callings the setter in target.

**Modified#2 – MessageTest.java**

```
package com.injection.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.ioc.beans.MessageWriter;

public class MessageWriterTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/ioc/common/application-context.xml"));
                MessageWriter messageWriter = factory.getBean("messageWriter",
                                MessageWriter.class);
                messageWriter.writeMessage("Welcome to Spring");
        }
}
```

Example shows how to inject IMessageProducer into MessageWriter using constructor injection.

In order to perform constructor injection, your target class MessageWriter instead of exposing a setter, should expose a constructor which should take IMessageProducer as parameter.

Along with this need to modify the "Spring beans configuration" to instruct to inject by calling constructor using <constructor-arg> tag rather than setter injection.

Below code fragment shows how to implement the same.

**Modified#2 - MessageWriter.java**

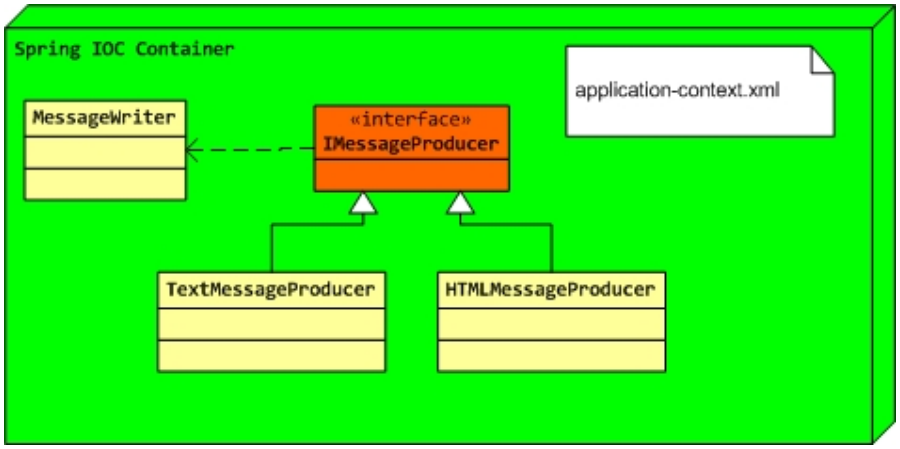
```
package com.ioc.beans;

public class MessageWriter {
   private IMessageProducer messageProducer;
   // constructor taking IMessageProducer parameter
   public MessageWriter(IMessageProducer messageProducer) {
      this.messageProducer = messageProducer;
   }

   public void writeMessage(String message) {
      String formattedData = messageProducer.formatMessage(message);
      System.out.println(formattedData);
   }
}
```

**Spring Beans configuration File (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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
        <bean id="textMessageProducer"
                class="com.ioc.beans.TextMessageProducer" />
        <bean id="htmlMessageProducer"
                class="com.ioc.beans.HTMLMessageProducer" />
        <bean id="messageWriter" class="com.ioc.beans.MessageWriter">
                <constructor-arg ref="htmlMessageProducer"/>
        </bean>
</beans>
```

If you observe the above examples it's clearly evident that our application components are loosely coupled and instead of we creating the objects and managing the dependencies, spring is doing this for us. So, as we specify the things in configuration it is easy to maintain and modify without touching the source of our application.

Below diagram shows the IOC process through Setter or Constructor injection



## 6.4 Constructor VS Setter Injection

As said above we have two types of Dependency Injection's, constructor injection and setter injection. We need to under what is the difference between constructor and setter before proceeding.

| Constructor Injection | Setter Injection |
|---|---|
| • At the time of creating your target class object, the dependent objects are available (can be accessed in the constructor of target class).<br>• In case of constructor injection all the dependent objects are mandatory to be injected, if you don't provide any of the dependent objects through <contructor-arg> tag the core container will detects and throws BeanCreationException.<br>• If classes have cyclic dependencies via constructor, these dependent beans cannot be configured through Constructor Injection. | • The dependent objects will not available while creating the target classes. After the target class has been instantiated, then the setter on target will be called to inject the dependent objects.<br>• In case of setter injection your dependent objects are optional to be injected. Even you don't provide the <property> tag while declaring the bean; the container will creates the Bean and initializes all the properties to their default.<br>• Cyclic dependencies are allowed in Setter Injection. |

## 6.5 Resolving Constructor Confusion

If a class contains overloaded constructors, and if you are trying to perform constructor injection, your constructor arguments resolution matching occurs using argument's type. If no potential ambiguity exists in the constructor arguments then the order in which those were declared in the configuration file will be mapped to the constructor arguments.

Let's consider a case where spring cannot resolve the constructor arguments directly.

**Robot.java**

```
package com.cc.beans;

public class Robot {
	private int id;
	private String name;

	public Robot(int id) {
		this.id = id;
	}

	public Robot(String name) {
		this.name = name;
	}

	public void showRobot() {
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
	}
}
```

In the above example the Robot class has two constructors one will take int and another takes string as parameter, if you configure this class as spring bean, you need to pass the argument for either of these constructors to instantiate as shown below.

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

	<bean id="robot" class="com.cc.beans.Robot">
		<constructor-arg value="10"/>
	</bean>

</beans>
```

The above configuration will makes a call to String argument constructor even you passed an integer value to it. This ambiguity rises as the "10" is compatible to String. To resolve this, you need to specify the type of value you are passing to it, so that it will detect the appropriate constructor in performing inject. Code shown below.

**Modified #1 - 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"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="robot" class="com.cc.beans.Robot">
		<constructor-arg value="10" type="int"/>
	</bean>

</beans>
```

Let us consider one more scenario where the same robot class contains two argument constructors as well as shown below.

**Modified - #2 Robot.java**

```
package com.cc.beans;

public class Robot {
	private int id;
	private String name;

	public Robot(int id, String name) {
		this.id = id;
		this.name = name;
	}

	public void showRobot() {
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
	}
}
```

If you try to configure this as spring bean, you need to configure the values in the same order of argument declaration. In case if the order mis-match then it will not be able to detect the relevant constructor. To resolve this you need to use index. Index lets you point the argument declaration to method parameters irrespective of the order in which those has been declared in configuration as shown below.

**Modified - #2 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"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="robot" class="com.cc.beans.Robot">
		<constructor-arg value="adf2" index="1"/>
		<constructor-arg value="10"/>
	</bean>
</beans>
```

## 6.6 Collection Injection

In spring you can inject four types of collections as dependent objects into your target classes. Those are List, Set, Map and Properties. Spring has provided tags that allow you to create these objects in declarations and allow you to inject into your target classes.

1) Injecting List

**Course.java**

```
package com.cdi.beans;

import java.util.List;
import java.util.Properties;
import java.util.Set;

public class Course {
    private List<String> subjects;

    public void setSubjects(List<String> subjects) {
		this.subjects = subjects;
    }
    public void showSubjects() {
		System.out.println("Subjects :");
		for (String s : subjects) {
			System.out.println(s);
		}
    }
}
```

In the above Course class we have List of subjects, while configuring the course class as a spring bean, we want to inject subjects list as well. In order to inject the list we need to use a <list> tag or <util:list> tag. Either using <list> or <util:list> has the same behavior, "util" namespace has been introduced from spring 2.0. The idea behind having the "util" namespace separately is to have namespace compartmentalization.

Below code snippet shows the configuration for injecting list.

**application-context.xml**

```
<bean id="bTechCS" class="com.cdi.beans.Course">
    <property name="subjects">
        <list value-type="java.lang.String">
            <value>C</value>
            <value>C++</value>
            <value>Java</value>
        </list>
    </property>
</bean>
```

2) Injecting Set

As you know the difference between list and set, list allows duplicates whereas set doesn't allow duplicates. You can inject set as dependent object using the tag <set>. Set has set of values, so the <set> tag contains <value> as child element, the same has been demonstrated in the below code.

**Course.java**

```
package com.cdi.beans;

import java.util.Set;

public class Course {
    private Set<String> faculties;

    public Course(Set<String> faculties) {
        this.faculties = faculties;
    }

    public void showFaculties() {
        System.out.println("Faculties :");
        for (String f : faculties) {
            System.out.println(f);
        }
    }
}
```

**application-context.xml**

```
<bean id="bTechCS" class="com.cdi.beans.Course">
    <constructor-arg>
        <set value-type="java.lang.String">
            <value>Mark</value>
            <value>John</value>
        </set>
    </constructor-arg>
</bean>
```

3) Injecting Map

Map is a collection which contains key and value pair. In case of Map the key can be any type and value can be any type. In order to create a map and inject into target class you need to use <map> tag. As map contains key and values the sub elements under it is <entry key=””><value></entry> tag. The below snippet shows the same.

**University.java**

```
package com.cdi.beans;

import java.util.Map;

public class University {
    private Map<String, Course> facultyCourseMap;

    public void setFacultyCourseMap(Map<String, Course> facultyCourseMap) {
        this.facultyCourseMap = facultyCourseMap;
    }

    public void showUniversityInfo() {
        System.out.println("University courses : ");
        for(String f : facultyCourseMap.keySet()) {
            System.out.println("********Course Info*****");
            Course c = facultyCourseMap.get(f);
            c.showSubjects();
            c.showFaculties();
            c.showFacultySubjects();
        }
    }
}
```

**application-context.xml**

```
<bean id="ou" class="com.cdi.beans.University">
    <property name="facultyCourseMap">
        <map key-type="java.lang.String" value-type="com.cdi.beans.Course">
            <entry key="mark">
                <ref bean="bTechCS"/>
            </entry>
            <entry key="john" value-ref="mca"/>
        </map>
    </property>
</bean>
```

4) Injecting Properties

Properties is also a Key and Value type collection, but the main difference between Map and Properties is Map can contain key and value as object type, but the Properties has key and value as string only.

In order to inject properties as dependent object, you need to use the tag <props>, this has sub elements <prop key="">value here</prop>.

Refer to the following example for the same.
**Course.java**

```
package com.cdi.beans;

import java.util.Properties;

public class Course {
    private Properties facultySubjects;

    public void setFacultySubjects(Properties facultySubjects) {
        this.facultySubjects = facultySubjects;
    }

    public void showFacultySubjects() {
        System.out.println("Faculty --> Subjects");
        for(Object o : facultySubjects.keySet()) {
            System.out.print(o + " --> ");
            System.out.println(facultySubjects.get(o));
        }
    }
}
```

**application-context.xml**

```
<bean id="mca" class="com.cdi.beans.Course">
   <property name="facultySubjects">
         <props>
               <prop key="Mark">C</prop>
               <prop key="John">S.E</prop>
         </props>
   </property>
</bean>
```

## 6.7 Bean Inheritance

Inheritance is the concept of reusing the existing functionality. In case of java you can inherit a class from an Interface or another class. When you inherit a class from another class, your child or derived class can you all of the functionalities of your base class.

When it comes to spring Bean inheritance, it talks about how to re-use the existing bean configuration instead of re-defining again. Let's consider a scenario where your class contains 10 attributes, if you want to configure it as spring bean, you need to inject values for 10 attributes via constructor or setter injection.

If we want to create 10 beans of that class, we need to configure for all the 10 beans the setter or constructor injection. In case if most of the attributes has same value, even then also we need to re-write the configuration. This leads to duplicate configuration declaration results in high amount of maintenance.

In order to avoid this you can declare the configuration in one bean which acts as parent bean. And all the remaining 9 bean declarations can inherit their declaration values from the parent bean, so that we don't need to repeatedly write the same configuration in all the child beans. In this way if we modify the attribute value in parent bean, it will automatically reflects in all its 9 child beans.

The child bean can override the inherited value of parent by re-declaring at the child level. Refer to the below snippet for the same.

**Car.java**

```
package com.bi.beans;

public class Car {
	private int id;
	private String name;
	private String engineType;
	private String engineModel;
	private String classType;

	public void setId(int id) {
		this.id = id;
	}

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

	public void setEngineType(String engineType) {
		this.engineType = engineType;
	}

	public void setEngineModel(String engineModel) {
		this.engineModel = engineModel;
	}

	public void setClassType(String classType) {
		this.classType = classType;
	}

	public void showCar() {
		System.out.println("Car Info");
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
		System.out.println("Engine Type : " + engineType);
		System.out.println("Engine Model : " + engineModel);
		System.out.println("Class Type : " + classType);
	}

}
```

**application-context.xml**

```
<bean id="baseCar" class="com.bi.beans.Car" abstract="true">
	<property name="engineType" value="Disel"/>
	<property name="engineModel" value="DDIS"/>
	<property name="classType" value="HatchBack"/>
</bean>

<bean id="swift" class="com.bi.beans.Car" parent="baseCar">
	<property name="id" value="23"/>
	<property name="name" value="Swift"/>
</bean>
```

In the above configuration we declared baseCar as abstract which means spring IOC container will not instantiate the object for the bean declaration. But it acts as a base bean from which its property values will be inherited to child beans.

## 6.8 Collection Merging

In spring 2.0, the container supports collection merging. In this your parent bean can declare a list as parent list. In your child bean you can declare a list with values; you can inherit the values of your parent list values into your child bean list values. As the list is merged with parent list values, this is called Collection Merging.

**Course.java**

```
package com.cm.beans;

import java.util.List;

public class Course {
	private List<String> subjects;

	public void setSubjects(List<String> subjects) {
		this.subjects = subjects;
	}

	public void showCourse() {
		System.out.println("Subjects :");
		for (String s : subjects) {
			System.out.println(s);
		}
	}
}
```

**application-context.xml**

```
<bean id="bTech1Yr1Sem" class="com.cm.beans.Course" abstract="true">
	<property name="subjects">
		<list>
			<value>c</value>
			<value>DMS</value>
		</list>
	</property>
</bean>

<bean id="bTechCS" class="com.cm.beans.Course" parent="bTech1Yr1Sem">
	<property name="subjects">
		<list merge="true">
			<value>S.E</value>
		</list>
	</property>
</bean>
```

## 6.9 Inner Beans

An inner bean is the concept similar to Inner classes in Java. As how you can create a class inside another class, you can inject a bean into another bean by declaring it inline. Below snippet shows the same.

**Motor.java**

```
package com.dc.beans;

public class Motor {
	private Engine engine;

	public void setEngine(Engine engine) {
		this.engine = engine;
	}

	public void run() {
		System.out.println("Running with engine : " + engine.getName());
	}
}
```

**Engine.java**

```
package com.dc.beans;

public class Engine {
	private int id;
	private String name;

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

**application-context.xml**

```
<bean id="motor" class="com.dc.beans.Motor" dependency-check="objects">
	<property name="engine">
		<bean class="com.dc.beans.Engine" dependency-check="simple">
			<property name="id" value="24"/>
			<property name="name" value="100 cc"/>
		</bean>
	</property>
</bean>
```

## 6.10 Using IDRef

The idref element is simply an error-proof way of passing the id of a bean into another bean. Let's take a scenario where a bean wants to retrieve another bean instead of injection.

If a bean wants to fetch another bean from factory, we need to pass id of the bean via constructor or setter injection into your target bean. If we pass the id as value, if your dependent bean id changes it has to be reflected in all the beans where it has been injected. If not your target bean tries to fetch the dependent bean with wrong id, which results in Null.

In order to avoid these configuration mis-matches, to pass the id of another bean we need to pass it as idref rather than simple value as shown below.

**BiCycle.java**

```
package com.idref.beans;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;

public class BiCycle {
	private String chainName;

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

	public void run() {
		BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("com/idref/common/application-context.xml"));
		Chain c = factory.getBean(chainName, Chain.class);
		System.out.println("I am running with BiCycle reference : " + bName);
		System.out.println("Running with Chain : " + c.getType());
	}

	public void setChainName(String chainName) {
		System.out.println("Setter");
		this.chainName = chainName;
	}
}
```

**Chain.java**

```
package com.idref.beans;

public class Chain {
	private String type;

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}

}
```

**application-context.xml**

```
<bean id="biCycle" class="com.idref.beans.BiCycle">
	<property name="chainName">
		<idref bean="ct1"/>
	</property>
</bean>

<bean id="ct1" class="com.idref.beans.Chain">
	<property name="type" value="t1"/>
</bean>
```

## 6.11 Bean Aliasing

In spring when you configure a class as Bean you will declare an id with which you want to retrieve it back from the container. Along with id you can attach multiple names to the beans, and these names acts as alias names with which you can look up the bean from the container.

Prior to spring 2.0 in order to declare multiple names you need to declare an "name" attribute at the bean tag level whose value contains beans names separated with ",".

Following code snippet shows the same.

**application-context.xml**

```
<bean id="robot" name="agent, machine" class="com.ba.beans.Robot"/>
```

You can retrieve the above bean with either robot or agent or machine names. You can even get all the names of the bean using factory.getAliases("onename").

In general bean aliasing is used for ease maintenance of the configuration.

In spring 2.0 a new tag has been introduced <alias> using which you can declare multiple names for the bean. The syntax is as follows

```
<bean id="robot" class="com.ba.beans.Robot"/>
<alias name="agent" bean="robot"/>
<alias name="machine" bean="robot"/>
```

## 6.12 Null String

The concept of Null string is how to pass Null Value for a Bean property. Let's consider a case where a class has attribute as String or other Object. We are trying to inject the value of this attribute using Constructor Injection.

In case of constructor injection the dependent object is mandatory to be injected in configuration. In case if the dependent object is not available, you can pass null for the dependent object in the target class constructor as shown below.

**Motor.java**

```
package com.un.beans;

public class Motor {
	private String id;

	public Motor(String id) {
		this.id = id;
	}
	public void showMotor() {
		System.out.println("Id : " + id);
	}
}
```

**application-context.xml**

```
<bean id="m1" class="com.un.beans.Motor">
	<constructor-arg>
		<null/>
	</constructor-arg>
</bean>
```

If you see the declaration, in order to pass null as value for the dependent object you need to use the tag <null/>.

## 6.13 Bean Scopes

In spring when you declare a class as a bean by default the bean will be created under singleton scope. Before understanding about scopes we need to understand what singleton class is and when to use it.

**What is singleton, when to use.**

When we create a class as singleton, it means we have only one instance of the class within the classloader.

We need to use singleton class in the below scenarios.

1) If a class has absolutely no state then declare those classes as singleton, as the class doesn't contains any attributes and it has only behavior's, calling the methods with object1 or object 2 doesn't makes any difference. So, instead of floating multiple objects in memory, we can have one object calling any methods of that class.

2) If a class has some state, but the state is read-only in nature then all the objects of that class sees the same state, so using the class behavior's with one object or n objects doesn't makes any difference, so we can use only one object of the class rather than multiple.

3) If a class has some state, but the state can be shared across multiple objects of the class. The state it contains is not only sharable but also it is very huge in nature, so instead of declaring multiple objects of that class we can allow to access the shared state through one object. But should allow the write/read access to the state via serialized order (synchronized fashion). For e.g.. all the cache classes has shared state which is shared across multiple objects of that class, but write/read operations to the cache data will be allowed in a synchronized manner.

In all the above scenarios we need to declare the class as Singleton. If a class is inverse of the above principles we should not declare the class as singleton rather should create multiple instances to access it.

In spring you can declare a bean with 5 different scopes as follows.

1) Singleton – by default every bean declared in the configuration file is defaulted to singleton (unless specified explicitly). This indicates when you try to refer the bean through injection or factory.getBean() the same bean instance will be returned from the core container.

2) Prototype – When we declare a bean scope as prototype this indicates every reference to the bean will return a unique instance of it.

3) Request – When we declare a bean scope as request, for every HTTPRequest a new bean instance will be injected

4) Session – For every new HttpSession, new bean instance will be injected.

5) Global Session – the globalsession scope has been removed from Spring 3.0. This used in Spring MVC Portlet framework where if you want to inject a new bean for a Portal Session you need to use this scope.

By the above it is clear that you can use request and session in case of web applications. So we will postpone the discussion on these till Spring MVC.

Let's example how to use singleton and prototype.

**DateUtil.java**

```
package com.bs.beans;

import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {
	public String formatDate(Date dt, String pattern) {
		String s = null;
		SimpleDateFormat sdf = new SimpleDateFormat(pattern);
		s = sdf.format(dt);
		return s;
	}
}
```

**application-context.xml**

```
<bean id="dateUtil" class="com.bs.beans.DateUtil" scope="prototype"/>
```

**BeanScopeTest.java**

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                          "com/bs/common/application-context.xml"));
DateUtil du1 = factory.getBean("dateUtil", DateUtil.class);
DateUtil du2 = factory.getBean("dateUtil", DateUtil.class);
System.out.println(du1 == du2);
```

As in the configuration we declare the dateUtil bean scope as prototype the comparision between du1 == du2 will returns false. If we set the scope as singleton it will yields to true.

## 6.14 Bean Autowiring

In spring when you want to inject one bean into another bean, we need to declare the dependencies between the beans using <property> or <constructor-arg> tag in the configuration file. This indicates we need to specify the dependencies between the beans and spring will reads the declarations and performs injection.

But when it comes to autowiring, instead of we declaring the dependencies we will instruct spring to automatically detect the dependencies and perform injection between them.

So, in order to do this we need to enable autowiring on the target bean into which the dependent has to be injected. You can enable autowiring in 4 modes.

1) byname – If you enable autowiring by name, spring will finds the attribute name which has setter on the target bean, and finds the bean in configuration whose name is matching with the attribute name and performs the injection by calling the setter. Following code demonstrates the same.

**Humpty.java**

```
package com.ba.beans;

public class Humpty {
	private Dumpty dumpty;

	public void setDumpty(Dumpty dumpty) {
		System.out.println("Setter");
		this.dumpty = dumpty;
	}

	public void showHumpty() {
		System.out.println("I am working with Dumpty : " + dumpty.getName());
	}
}
```

**Dumpty.java**

```
package com.ba.beans;

public class Dumpty {
	private String name;

	public String getName() {
		return name;
	}

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

**application-context.xml**

```
<bean id="humpty" class="com.ba.beans.Humpty" autowire="byName"/>

<bean id="dumpty" class="com.ba.beans.Dumpty">
	<property name="name" value="Dumpty12"/>
</bean>
```

If you see the above configuration, on the humpty we enabled autowire byName. With this configuration, it will tries to find the Humpty beans attributes which has setters, with that attribute name "dumpty" it will tries to find a relevant bean with the same name "dumpty" as the bean is available it will injects the dumpty bean into Humty attribute.

2) byType – If we enable autowire byType, if will find the attributes type in the target class and tries to identify a bean from the configuration file of the same type and then injects into target class by calling setter on top of it. Below configuration demonstrates the same.

**application-context.xml**

```
<bean id="humpty" class="com.ba.beans.Humpty" autowire="constructor"/>

<bean id="dumpty12" class="com.ba.beans.Dumpty">
	<property name="name" value="Dumpty12"/>
</bean>
```

In the above case the bean attribute name is "dumpty", and in the configuration the bean name is "dumpty12" even though the names are not matching still the dumpty12 bean will be injected into humpty. Because the type of the attribute and the bean type is matching.

Note:- if multiple bean declarations of the same type is found it will throw an ambiguity error without instantiating the core container.

3) Constructor – If we enable autowire in constructor mode, now it will tries to find a bean whose class type is same as constructor parameter type, if a matching constructor is found it will passes the bean reference to its constructor and performs injection. This means it is similar to byType but instead of calling setter it will call constructor to perform injection.

**Humpty.java**

```
package com.ba.beans;

public class Humpty {
	private Dumpty dumpty;

	public Humpty() {
		super();
	}
	public Humpty(Dumpty dumpty) {
		System.out.println("Constructor");
		this.dumpty = dumpty;
	}

	public void showHumpty() {
		System.out.println("I am working with Dumpty : " + dumpty.getName());
	}
}
```

There is no change in configuration apart from declaring on the bean autowire=”constructor”

4) Autodetect – this has been removed from spring 3.0 onwards as it is quite confusing. In this it will tries to perform injection by finding a relevant constructor by type if not found then it will finds the setter by type and performs injection.

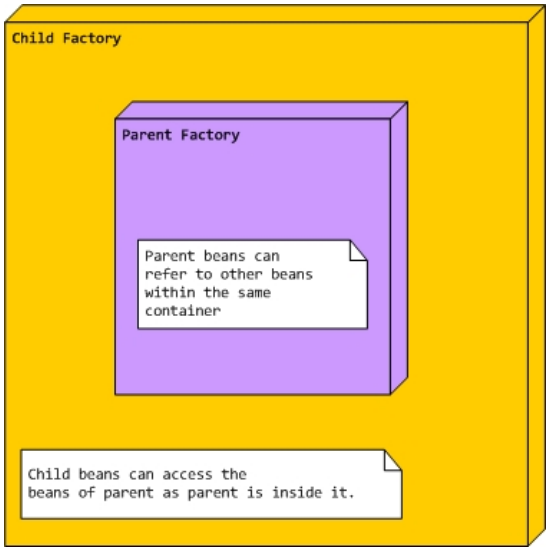
Drawback with autowiring – The problem with autowiring is we don’t have control over which beans has to get injected into what, so it is least recommended to use autowiring for a large applications. For pilot projects where RAPID application development is needed we use autowiring.

## 6.15 Nested BeanFactories

If we have two bean factories in an application, we can nest one bean factory into another to allow the beans in one bean factory to refer to the beans of other factory. In this we declare one bean factory as parent bean factory and will declare the other as child.

This is similar to the concept of base class and derived classes. Derived class can access the attributes of base class, but base class cannot access the attributes of derived class.

In the same way child bean factory beans can refer to the parent bean factory beans. But parent bean factory beans cannot refer to child beans declared in child factory. Pictorial representation of it is shown below.




Below example shows how to use it.

**EMICalculator.java**

```
package com.nbf.beans;

public class EMICalculator {
	public float compute(long principal, float rateOfInterest, int years) {
		return 343.34f;
	}
}
```

**CustomerLoanApprover.java**

```
package com.nbf.beans;

public class CustomerLoanApprover {
	private EMICalculator emiCalculator;

	public void approve(double grossSalary, long principalAmount, int years) {
		float emi = emiCalculator.compute(principalAmount, 13.5f, years);
		System.out.println("Emi : " + emi);
		if (emi > 0) {
			System.out.println("Approved");
		} else {
			System.out.println("Rejected");
		}
	}

	public void setEmiCalculator(EMICalculator emiCalculator) {
		this.emiCalculator = emiCalculator;
	}

}
```

If the EMICalculator class has been declared in one configuration file and CustomerLoanApprover class has been declared in second configuration file. In order to inject EMICalculator into CustomerLoanApprover class we need to nest their factories as shown below.

**loan-beans.xml**

```
<bean id="emiCalculator" class="com.nbf.beans.EMICalculator"/>
```

**customer-beans.xml**

```
<bean id="customerLoanApprover" class="com.nbf.beans.CustomerLoanApprover">
	<property name="emiCalculator">
		<ref parent="emiCalculator"/>
	</property>
</bean>
```

In the above configuration in order for your bean declaration to refer to parent beans, it has to use the tag <ref parent = “”/>. Apart from parent attribute it has local which indicates refer to the local bean. Along with it we have bean attribute as well, which indicates look in local if not found the search in parent factory and perform injection.

**NBFTest.java**

```
public static void main(String[] args) {
        BeanFactory pf = new XmlBeanFactory(new ClassPathResource(
                        "com/nbf/common/loan-beans.xml"));
        BeanFactory cf = new XmlBeanFactory(new ClassPathResource(
                        "com/nbf/common/customer-beans.xml"), pf);

        CustomerLoanApprover cla = cf.getBean("customerLoanApprover",
                        CustomerLoanApprover.class);
        cla.approve(3423.3f, 3535, 324);
}
```

If you observe the above code while creating the "cf" factory we passed the reference of "pf" to create it. This indicates "cf" factory has been nested from "pf".

This completes the Spring Core basic concepts and enables us to proceed for advanced spring core concepts.

# 7 Spring Core (Advanced)

## 7.1 Using P & C – Namespace

If we want to perform setter injection on a spring bean we need to use <property> tag. Instead of writing length <property> tag declaration under the <bean> tag, we can replace with short form of representing the same with p-namespace.

In order to use the p-namespace, you first need to import the "http://www.springframework.org/schema/p" namespace in the spring bean configuration file. Once you have imported it, you have to write the attribute at the <bean> tag level to perform the injection as **p:propertyname="value"** or **p:propertyname-ref="refbean"**.

C-Namespace has been introduced in spring 3.1.1, in order to perform constructor injection we need to use <constructor-arg> tag. Instead of writing the length <constructor-arg> tag, we can replace it with c-namespace. The syntax for writing the C-Namespace is **c:argument="value"** or **c:-argument-ref="refbean"**

**Course.java**

```
package com.pnamespace.beans;

public class Course {
	private int id;
	private String name;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

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

}
```

**Person.java**

```
package com.pnamespace.beans;

public class Person {
	private Course course;

	public Person(Course course) {
		this.course = course;
	}

	public void whichCourse() {
		System.out.println("Course : " + course.getName());
	}
}
```

**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:c="http://www.springframework.org/schema/c"
	xmlns:p="http://www.springframework.org/schema/p"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="course" class="com.pnamespace.beans.Course" p:id="34"
p:name="Java"/>
	<bean id="person" class="com.pnamespace.beans.Person" c:course-
ref="course"/>
</beans>
```

## 7.2 Dependency Check

In case of constructor injection, all the dependent objects are mandatory to be passed via <constructor-arg> tag while declaring the bean, but in case of setter injection the dependent objects are not mandatory to be injected. So, if want setter properties mandatory like similar to constructor properties then we need to use Dependency Check.

Dependency check has been removed from Spring 2.5 and is available prior to 2.5. From spring 2.5 it has been replaced with @Required annotation, we will discuss about it spring annotations support.

In order to perform the mandatory check on setter properties you need to enable dependency check. In order to enable dependency check you have to write the attribute **dependency-check="mode"**. Dependency check can be enabled in three modes.

1) Simple – when you turn the dependency check in simple mode, it will check all the primitive attributes of your bean (attributes contains setters) whether those has been configured with values in configuration file. If any primitive attribute is not configured with <property> or p-namespace in configuration automatically the core container will detects and throws error without creating the container.

2) Object- when you turn on the dependency check in object mode, it will check all the Object type attributes on your target class whether those has been configured the property injection in configuration, if not will throw exception as said above.

3) all – when you turn on the dependency check as all mode, it will check for both simple and objects types of your class attributes, if those values are not injected through configuration it will throw exception.

Refer the example for the same.

**Engine.java**

```
package com.dc.beans;

public class Engine {
	private int id;
	private String name;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

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

**Motor.java**

```
package com.dc.beans;

public class Motor {
	private Engine engine;

	public void setEngine(Engine engine) {
		this.engine = engine;
	}

	public void run() {
		System.out.println("Running with engine : " + engine.getName());
	}
}
```

**application-context.xml**

```
<bean id="engine" class="com.dc.beans.Engine" dependency-check="simple">
	<property name="id" value="24"/>
	<property name="name" value="100 cc"/>
</bean>

<bean id="motor" class="com.dc.beans.Motor" dependency-check="objects">
	<property name="engine" ref="engine"/>
</bean>
```

The main drawback with dependency check is you don't have control on which attribute should be made as mandatory and which is optional, either you can the dependency check at simple level or object or all.

## 7.3 Depends-On

Till now we are talking about how to manage dependencies between beans using another bean. When it comes to depends-on it doesn't talks about managing the dependencies between dependent beans, instead it talks about the creational dependencies of the beans.

If we have two beans say Employee and CacheManager, Employee uses some data in the Cache which has been loaded by CacheManager. Now Employee is dependent on Cache, but not on CacheManager. But only the CacheManager knows how to create the Cache. This indicates before the Employee gets instantiated first your cache manager should get created then only your employee bean can use the cache that has been built by the CacheManager. So in order to express such creational dependencies we need to use depends-on attribute at the <bean> tag as shown below.

**Cache.java**

```
package com.dpo.cache;

import java.util.HashMap;
import java.util.Map;

public class Cache {
	private static Cache _instance;
	private Map<String, String> data;

	private Cache(Map<String, String> data) {
		this.data = data;
	}

	private Cache() {

	}

	public synchronized static void load(Map<String, String> data) {
		if (_instance == null) {
			_instance = new Cache(data);
		}
	}

	public synchronized static Cache getInstance()
			throws IllegalAccessException {
		if (_instance == null) {
			throw new IllegalAccessException("Cache has not been initialized");
		}
		return _instance;
	}

	public synchronized String get(String key) {
		return data.get(key);
	}

	public synchronized void put(String key, String value) {
		data.put(key, value);
	}
}
```

**CacheManager.java**

```
package com.dpo.cache;

import com.dpo.accessor.DBDataAccessor;

public class CacheManager {

	public CacheManager() {
		// load the cache
		initialize();
	}

	public void initialize() {
		DBDataAccessor accessor = new DBDataAccessor();
		Cache.load(accessor.getData());
	}
}
```

**DBAccessor.java**

```
package com.dpo.accessor;

import java.util.HashMap;
import java.util.Map;

public class DBDataAccessor {
	public Map<String, String> getData() {
		Map<String, String> data = new HashMap<String, String>();

		// write db logic
		data.put("IND", "1");
		data.put("US", "2");

		return data;
	}
}
```

**SearchEmployee.java**

```
package com.dpo.beans;

import com.dpo.cache.Cache;

public class SearchEmployee {

	public void search(String countryNm) throws IllegalAccessException {
		Cache cache = Cache.getInstance();
		System.out.println("Country ID : " + cache.get(countryNm));
	}
}
```

**application-context.xml**

```
<bean id="searchEmployee" class="com.dpo.beans.SearchEmployee" depends-on="cacheManager"/>
<bean id="cacheManager" class="com.dpo.cache.CacheManager"/>
```

In the above configuration, irrespective of order in which you declared searchEmployee and cacheManager beans first your cacheManager will be created first and then only your searchEmployee will be created.

## 7.4 Bean Lifecycle

Bean lifecycle allows a way to provide spring beans to perform initialization or disposable operations.

If you consider a Servlet, J2EE containers will call the init and destroy methods after creating the servlet object and before removing it from the web container respectively. In the Servlet init method, developer has to write the initialization logic to initialize the Servlet and destroy method he needs to write the code for releasing the resources to facilitate the destruction process.

If you consider a POJO in Java, initialization logic has to be written in the constructor and resource release logic should be written in the finalize method.

In the same way, for spring beans also, the core container provides extension hooks to initialize its state and should facilitate to release the resources held be spring bean. Unlike your normal pojo or servlet lifecycle, spring bean lifecycle would be slightly different. In spring you can consider a bean has been completely constructor only after the constructor, setter or any other injection or performed and all its dependents are acquired. So if we write the initialization logic in constructor for a spring bean, only the dependents that are injected via constructor are available but the dependents that are injected via setter will not be available in the constructor.

So, we need a unified place where after all the dependents are available we want to perform initialization logic and before the container removes the bean, we want to perform destruction process.

In order to do this spring provides three ways to work with Bean Lifecycle.

1) Declarative approach
2) Programmatic approach
3) Annotation based approach

### 7.4.1 Declarative approach

In the declarative approach you will declare the init and destroy methods of a bean in spring beans configuration file. In the bean class declare the methods whose signature must be public and should have return type as void and should not take any parameters, any method which follows this signature can be used as lifecycle methods.

After writing the methods in your bean, you need to declare those methods as lifecycle methods in spring configuration file, at the <bean> tag level, you need to declare two attributes **init-method = "methodname"** and **destroy-method = "destroymethod"**.

IOC container after creating the bean (this includes after all injections); it will automatically calls the init method to perform the initialization. But spring IOC container cannot automatically calls the destroy-method of the bean class, because in spring core application IOC container will not be able to judge when the bean is available for garbage collection or how many references of this bean is held by other beans.

In order to invoke the destroy-method on the bean class you need to explicitly call on the ConfigurableListableBeanFactory, destroySingleton or destroyScopedBean("beanscope") method, so that IOC container delegates the call to all the beans destroy-methods in that IOC container.

Refer to the below example on how to work with declarative approach.

**Robot.java**

```
package com.blc.beans;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Robot {
  private String name;
  private SensorDriver driver;

  public Robot(SensorDriver driver) {
        this.driver = driver;
  }

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

  public void startup() {
        System.out.println("Driver Type : " + driver.getType());
        System.out.println("Name : " + name);
  }

  public void release() {
        System.out.println("releasing resources....");
  }

}
```

**SensorDriver.java**

```
package com.blc.beans;

public class SensorDriver {
  private String type;

  public String getType() {
        return type;
  }

  public void setType(String type) {
        this.type = type;
  }

}
```

**application-context.xml**

```
<bean id="robot" class="com.blc.beans.Robot" init-method="startup" destroy-
method="release">
	<constructor-arg ref="driver"/>
	<property name="name" value="Robot 1"/>
</bean>

<bean id="driver" class="com.blc.beans.SensorDriver">
	<property name="type" value="IR"/>
</bean>
```

**BLCTest.java**

```
package com.blc.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.blc.beans.Robot;
import com.blc.beans.ShutdownHookThread;

public class BLCTest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/blc/common/application-context.xml"));
		Robot robot = factory.getBean("robot", Robot.class);
		((ConfigurableListableBeanFactory)factory).destroySingletons();
	}
}
```

### 7.4.2 Programmatic approach

The problem with declarative approach is developer has provide the init and destroy method declarations for each bean of that class in configuration, if he misses for atleast one bean, then the initialization or destruction will not happen. It would be tough to maintain the configuration, lets say if the init or destroy method name changed in class, all the references to those methods in the configuration has to be modified to match with class method names.

In order to avoid the problems with this spring has provided programmatic approach. In this the bean class has to implement from two interfaces InitializingBean interface to handle initialization process and DisposableBean interface to handle destruction process and should override afterPropertiesSet and destroy methods respectively.

Now the developer don't need to declare these methods as init-method or destroy-method rather the core container while creating these beans will detects

these beans as InitializingBean type or DisposableBean type and calls the afterPropertiesSet and destroy method automatically.

The Robot.java has been modified to depict the same.

**Modified# 1 - Robot.java**

```
package com.blc.beans;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Robot implements InitializingBean, DisposableBean {
  private String name;
  private SensorDriver driver;

  public Robot(SensorDriver driver) {
        this.driver = driver;
  }

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

  @Override
  public void destroy() throws Exception {
        System.out.println("From destroy()");
        System.out.println("destroying dependents....");
  }

  @Override
  public void afterPropertiesSet() throws Exception {
        System.out.println("From afterPropertiesSet()");
        System.out.println("Driver Type : " + driver.getType());
        System.out.println("Name : " + name);
  }
}
```

We will discuss about the annotation approach while we are dealing with annotations.

## 7.5 Aware Interfaces

If a spring bean needs certain infrastructure dependency, it would declare to the container through Aware Interfaces. For example if a spring beans wants to access its factory or application context through which it was created, it can access it from BeanFactoryAware or ApplicationContextAware interfaces. Along with this if a bean wants to access its name with which it was configured in the configuration it can implement BeanNameAware. When a bean implements an aware interface, it has to override the method that is declared in that interface to get access to its infrastructure like factory, application context or bean name.

There are lot of aware interfaces that are provided in spring like BeanFactoryAware, BeanNameAware, ApplicationContextAware, ServletConfigAware, ServletContextAware, ApplicationEventPublisherAware etc.

Let's modify the BeanLifecycle example to register a shutDownHook which will automatically calls the destroySingleton method on BeanFactory, which in turn delegates the call to destroy methods on beans.

**ShutdownHook.java**

```
package com.blc.beans;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;

public class ShutdownHookThread implements Runnable, BeanFactoryAware {
	private BeanFactory factory;
	@Override
	public void run() {
		((ConfigurableListableBeanFactory)factory).destroySingletons();
	}
	@Override
	public void setBeanFactory(BeanFactory factory) throws BeansException {
		this.factory = factory;
	}
}
```

**BLCTest.java**

```
package com.blc.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.blc.beans.Robot;
import com.blc.beans.ShutdownHookThread;

public class BLCTest {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/blc/common/application-context.xml"));
		Runtime r = Runtime.getRuntime();
		ShutdownHookThread sdh = factory.getBean("shutDownHook",
				ShutdownHookThread.class);
		r.addShutdownHook(new Thread(sdh));
		Robot robot = factory.getBean("robot", Robot.class);
	}

}
```

**application-context.xml**

```
<bean id="shutDownHook" class="com.blc.beans.ShutdownHookThread"/>
```

In the above code we have created a class ShutdownHook which implements from BeanFactoryAware, which means the core container after creating the shutdown hook bean that has been declared in the configuration, it check whether the Bean is of BeanFactoryAware and then injects the BeanFactory by calling the setBeanFactory method that is declared in the interface. In this way the bean gets access to the Factor though which it has been created.

## 7.6 Static Factory Method

When you configure a class as spring bean, IOC container will instantiate the bean by calling the new operator on it. But in java not all the classes can be instantiated out of new operator. For example Singleton classes cannot be instantiated using new operator, rather you need to call the static factory method that has been exposed by the class to create object of that class.

So, for the classes which cannot be created out of new operator, those which has to be created by calling the static method that has been exposed in the class itself, you need to use static factory injection technic.

In this, after configuring the class as a bean, at the <bean> tag level you need to declare an attribute factory-method = "factorymethod" which is responsible for creating the object of that class. For example an alarm class needs an Calendar as dependent object, but java.util.Calendar class object cannot instantiated using new operation we need to call the getInstance() static method that is exposed in that class to get object of that class. Following example depicts the same.

**Alarm.java**

```
package com.sf.beans;

import java.util.Calendar;

public class Alarm {
	private Calendar time;

	public void setTime(Calendar time) {
		this.time = time;
	}
	public void ring() {
		System.out.println("Rining at : " + time.getTime());
	}
}
```

**application-context.xml**

```
<bean id="cal" class="java.util.Calendar" factory-method="getInstance" />

<bean id="alarm" class="com.sf.beans.Alarm">
	<property name="time" ref="cal" />
</bean>
```

## 7.7 Instance Factory Method

As said above not all class objects cannot be created out of new operator, few can be constructed by calling factory method on the class, but few objects can be constructed by calling methods on other classes.

If you consider EJB's are any other distributed objects, those cannot be constructor out of new operator or a static method. In order to create an EJB object you need to look up the reference of home object in the JNDI and then you need to call the create() method on it to get the EJB Remote object. This indicates that creation of object involves some sequence of steps or operations to be performed, so these steps are coded in a separate class methods and generally those classes are called ServiceLocator's, these are the classes who knows how to instantiate the objects of other classes.

So, in order to create such type of objects you need to instantiate the ServiceLocator or helper class on which in-turn you need to call the factory method to create our class object. Instance Factory method injection allows you to instantiate classes object by calling method on other (ServiceLocator) class. For this you need to declare the (Service Locator) class on which you call the factory method as spring bean, along with this configure the (Target) bean which has to be created by the Service Locator and declare on the <bean> tag factory-bean is your service locator bean and factory-method is your method in service locator class which will creates your target bean, which is shown below.

**GoogleMapRenderer.java**

```
package com.inf.beans;

public class GoogleMapRenderer {
	private MapEngine mapEngine;

	public void render(String source, String destination) {
		String directions[] = mapEngine.getDirections(source, destination);
		System.out.println("Directions :");
		for (String d : directions) {
			System.out.println(d);
		}
	}

	public void setMapEngine(MapEngine mapEngine) {
		this.mapEngine = mapEngine;
	}
}
```

**MapEngine.java**

```
package com.inf.beans;

public interface MapEngine {
	String[] getDirections(String source, String destination);
}
```

**IndiaMapEngine.java**

```
package com.inf.beans;

public class IndiaMapEngine implements MapEngine {

	@Override
	public String[] getDirections(String source, String destination) {
		return new String[] { "a", "b", "c" };
	}
}
```

**USMapEngine.java**

```
package com.inf.beans;

public class USMapEngine implements MapEngine {

	@Override
	public String[] getDirections(String source, String destination) {
		return new String[] { "x", "y", "z" };
	}
}
```

**MapEngineLocator.java**

```
package com.inf.beans;

public class MapEngineLocator {

	public MapEngine getIndiaMapEngine() {
		return new IndiaMapEngine();
	}

	public MapEngine getUSMapEngine() {
		return new USMapEngine();
	}
}
```

**application-context.xml**

```
<bean id="mapEngineLocator" class="com.inf.beans.MapEngineLocator" />

<bean id="indiaMapEngine" factory-bean="mapEngineLocator"
        factory-method="getMapEngine" />
<bean id="usMapEngine" factory-bean="mapEngineLocator"
        factory-method="getMapEngine"/>

<bean id="googleMapRenderer" class="com.inf.beans.GoogleMapRenderer">
        <property name="mapEngine" ref="indiaMapEngine" />
</bean>
```

## 7.8 Factory Bean

The problem with static factory method and instance factory method injection is the factory method you configure should take zero arguments. In case if the factory method of your class takes parameters, it cannot be declared as factory-method in the configuration file.

So, if you bean creation process is quiet complex where you need to pass some inputs to factory methods to create object of it then instead of using factory method or instance factory method we need to use Factory Beans.

In order to work with Factory Beans, you need to create a class which should implement from FactoryBean interface and need to override three methods getObject, getObjectType and isSingleton. In the getObject method you need to return the Object of the class you want to create, in the getObjectType you need to return the class type of the Object and isSingleton indicates the object you want to create through the Factory is singleton or prototype.

While creating the object of the class, if you have to pass parameters to the factory method, you need to get all those values as setter or constructor injection into Factory Bean class. Once all the dependent values are injected, you need to create your object in afterPropertiesSet method and should return the object in getObject method. This is shown in the below example, let's us modify the MapEngineLocator class to have a Factory method which takes parameter as type.

Now we cannot configure the MapEngineLocator bean as Instance Factory method because the factory method takes argument as type. In order to inject MapEngine we need to create a MapEngineFactory who know how to instantiate MapEngine Object by calling the factory method on MapEngineLocator class.

**MapEngineLocator.java**

```
package com.inf.beans;

public class MapEngineLocator {

	public MapEngine getMapEngine(String type) {
		if ("India".equals(type)) {
			return new IndiaMapEngine();
		} else {
			return new USMapEngine();
		}
	}
}
```

In the above code factory method takes argument, so we cannot inject the MapEngine by configuring MapEngineLocator as Instance Factory Injection.

**MapEngineFactory.java**

```
package com.inf.beans;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

public class MapEngineFactory implements FactoryBean<MapEngine>,
		InitializingBean {
	private String type;
	private MapEngine engine;
	@Override
	public void afterPropertiesSet() throws Exception {
		engine = new MapEngineLocator().getMapEngine(type);
	}
	public void setType(String type) {
		this.type = type;
	}
	@Override
	public MapEngine getObject() throws Exception {
		return engine;
	}
	@Override
	public Class<?> getObjectType() {
		return MapEngine.class;
	}
	@Override
	public boolean isSingleton() {
		return true;
	}

}
```

**application-context.xml**

```
<bean id="mapEngineFactory" class="com.inf.beans.MapEngineFactory">
        <property name="type" value="US"/>
</bean>
<bean id="googleMapRenderer" class="com.inf.beans.GoogleMapRenderer">
        <property name="mapEngine" ref="mapEngineFactory" />
</bean>
```

In the above configuration we are injecting mapEngineFactory into GoogleMapRenderer, while spring instantiating the googleMapRenderer bean, while injecting the mapEngineFactory bean via setter, instead of injecting the FactorBean, it will calls getObject method on the mapEngineFactory and the returned object will be injected into the googleMapRenderer bean.

## 7.9 Method Replacement

In spring you can replace any method on an arbitrary bean without touching the existing logic in it. This approach has numerous number of advantages, firstly your original method is untouched which means if you want to roll back to original code always you have an option for it. Second the original method logic will be replaced with a new logic by declaring it in spring configuration file. As the things are driven through configuration at any point of time if you want to come back to original logic, you just need to modify the configuration rather than code, so this doesn't require a build cycle.

For replacing a method on a bean the method should not be final and should not be static. In order to replace the method logic, you need to write a new class which implement from MethodReplacer interface and should override reimplement method.

The reimplement method takes targetObject, java.lang.Method and Object[] arguments as parameters. These parameters are the values with which you called the original method.

Let's say your original method on your bean is add(int a, int b) as parameters, now your reimplement method will be passed the object with which you invoked add as first parameter and add as Method in second parameter and the third parameter Object[] args contains two integers a and b resply.

Now you need to instruct the core container asking it to replace the add method with reimplement method logic, which indicates whenever I call add method on my bean, instead of calling add, call reimplement method by passing all the parameters as Object[] args. This would be done through declaration.

In the declaration after configuring your MathCalculator bean which contains add method as bean, you need to declare the tag <replaced-method name="add" and replacer = "replacementclass as bean"/>

The same is explained in the below example.

**PlanFinder.java**

```
package com.mr.beans;

public class PlanFinder {
	public String findPlan(String ageGroup, String coverageType,
			boolean isMilitaryPersonel) {
		return "Jeevan Saral";
	}
}
```

**FindPlanReplacer.java**

```
package com.mr.beans;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class FindPlanReplacer implements MethodReplacer {

	@Override
	public Object reimplement(Object target, Method method, Object[] args)
			throws Throwable {

		// perform a check to understand which method you are replacing
		if (method.getName().equals("findPlan")) {
			String ageGroup = null;
			String coverageType = null;
			boolean isMilitaryPersonel = false;

			ageGroup = (String) args[0];
			coverageType = (String) args[1];
			isMilitaryPersonel = (Boolean) args[2];

			System.out.println("ageGroup : " + ageGroup);
			System.out.println("CoverageType : " + coverageType);
			System.out.println("isMilitaryPersonnel : " + isMilitaryPersonel);

			// perform new computation
		}
		return "Jeevan Anand";
	}

}
```

**application-context.xml**

```
<bean id="findPlanReplacer" class="com.mr.beans.FindPlanReplacer"/>

<bean id="planFinder" class="com.mr.beans.PlanFinder">
        <replaced-method name="findPlan" replacer="findPlanReplacer">
                <arg-type>java.lang.String</arg-type>
                <arg-type>java.lang.String</arg-type>
                <arg-type>java.lang.Boolean</arg-type>
        </replaced-method>
</bean>
```

Spring will replaces the findPlan method logic with reimplement method of findPlanReplacer at runtime by generating a new class extending from PlanFinder class and override the method findPlan with the logic that is there in reimplement method.

In order to replace the logic and to generate new class spring uses CGLIB runtime byte code generation libraries.

Note: - spring recommends writing one replacer for one method and is not recommended to replace multiple methods with one replacer.

## 7.10 Lookup Method Injection

In spring when a non-singleton bean is getting injected into a singleton bean the result would be always singleton. For example if LoanApprover is a singleton class and LoanInfo is a non-singleton class. If we inject LoanInfo into LoanApprover class as shown below.

```
<bean id=”loanApprover” class=”com.lmi.beans.LoanApprover”>
        <property name=”loanInfo” ref=”loanInfo”/>
</bean>
<bean id=”loanInfo” class=”com.lmi.beans.LoanInfo” scope=”prototype”/>
```

As the loanApprover is a singleton class, only one object of LoanApprover will be created by core container and hence LoanInfo bean will be injected once via setter injection into LoanApprover class. This leads to LoanInfo as singleton.

Ideally speaking the above design is wrong, because if we want to declare a class as singleton, it should not contain any state. As the LoanApprover is using the LoanInfo, instead of injecting LoadInfo via setter or constructor injection, LoanApprover has to fetch the LoanInfo from the container and should use and dispose it.

The below table depicts the matrix of combinations of single-ton and non-singleton.

| Target Bean | Dependent Bean | Result |
|---|---|---|
| Singleton | Singleton | ✓ |
| Singleton | Non-Singleton | x<br>(Should not use setter or constructor, use lookup method injection) |
| Non-Singleton | Singleton | ✓ |
| Non-Singleton | Non-Singleton | ✓ |

Let's try to understand this better by taking one more example. The Web container upon receiving a request from the client, it will try to process the request by creating a new Object of RequestHandler class and populates the request data to RequestHandler.

If you observer the above scenario, here the Web container is a singleton class and RequestHandler is a non-singleton class, for every incoming request to the container it has to create a new RequestHandler class object and populate the request info and then should handover the control to process the request. So, the Web container will fetch the RequestHandler (so that each lookup of request handler will returns new object) class object upon receiving the request and populates data to process.

The Web Container class can fetch the Object of RequestHandler by implement BeanFactoryAware and can call getBean method to get the RequestHandler object shown below.

```
package com.lmi.beans;

public class WebContainer implements BeanFactoryAware {
	private BeanFactory factory;

	public void process(String data) {
		RequestHandler rh = factory.getBean("requestHandler",
RequestHandler.class);
		rh.setData(data);
		rh.handle();
	}

	public void setBeanFactory(BeanFactory factory) {
		this.factory = factory;
	}
}
```

In the above code we have used factory.getBean("requestHandler"), this indicates that we have hardcoded the logical name of the bean in our code, so that our code is tightly coupled with that specific bean. Second issue with that code is we have implemented our code from spring specific interface which indicates we loose the benefit of non-invasive feature of spring.

To avoid the above problems we need to use Lookup Method Injection. In this your class doesn't need to implement from any spring specific class or interface. You don't need to code for getting the reference of dependent object. Instead you declare to spring asking it to write the code of getting the dependent object through configuration, so that your code is loosely coupled with a spring bean. This is shown in the below example.

**WebContainer.java**

```
package com.lmi.beans;

abstract public class WebContainer {

	public void process(String data) {
		RequestHandler rh = getRequestHandler();
		rh.setData(data);
		rh.handle();
	}

	abstract public RequestHandler getRequestHandler();
}
```

**RequestHandler.java**

```
package com.lmi.beans;

public class RequestHandler {
	private String data;

	public void setData(String data) {
		this.data = data;
	}

	public void handle() {
		System.out.println("Processing request with data : " + data);
	}
}
```

In the above code we declared a method getRequestHandler() as a abstract method which returns RequestHandler object. We don't know how to write the logic in this method to get the RequestHandler object, so we declared this as abstract asking spring to implement this method for us. We will tell spring to implement the getRequestHandler method in configuration as shown below.

**application-context.xml**

```
<bean id="requestHandler" class="com.lmi.beans.RequestHandler"
scope="prototype"/>
<bean id="webContainer" class="com.lmi.beans.WebContainer">
        <lookup-method name="getRequestHandler" bean="requestHandler"/>
</bean>
```

In the above declaration we declared lookup method tag in WebContainer bean stating spring to implement the method getRequestHandler to return the requestHandler bean.

So your code has to call the getRequestHandler method to get the object of it. Spring will generates a class at runtime by using CGLib libraries and overrides this method to return the RequestHandler object.

## 7.11 Property Editors

In spring when you configure a class as spring bean, you can inject all the dependent attributes of the bean via setter or constructor injection. The attributes that you want to inject via spring injection could be of any type like int, float, long, any arbitrary object, String[], File etc.

Spring when you configure an attribute value using <property> tag, the value will be converted into class attribute type by using the property editor and then injects into the bean.

So there are lot of in-built property editors that are available in spring which converts the string values you configured in the configuration file to the target bean attribute types. For example when you configure a file path as string in <property> or <constructor-arg> tag it will be injected as a File object into the target class attribute using the file path. So these types of conversions will be done using PropertyEditor.

As we understood the purpose of PropertyEditor, let's try to understand how to create our own custom property editor by taking an example.

**ComplexNumber.java**

```
package com.pe.beans;
public class ComplexNumber {
	private int base;
	private int expo;
	public ComplexNumber(int base, int expo) {
		this.base = base;
		this.expo = expo;
	}
	public int getBase() {
		return base;
	}
	public void setBase(int base) {
		this.base = base;
	}
	public int getExpo() {
		return expo;
	}
	public void setExpo(int expo) {
		this.expo = expo;
	}
}
```

**MathCalculator.java**

```
package com.pe.beans;

public class MathCalculator {
	private ComplexNumber complexNumber;

	public void setComplexNumber(ComplexNumber complexNumber) {
		this.complexNumber = complexNumber;
	}

	public void calculate() {
		System.out.println("Calculating with complex number base : "
				+ complexNumber.getBase() + " expo : "
				+ complexNumber.getExpo());
	}
}
```

In order to inject ComplexNumber into MathCalculator, the configuration looks as follows.

**application-context.xml**

```
<bean id="complexNumber" class="com.pe.beans.ComplexNumber">
	<property name="base" value="24"/>
	<property name="expo" value="34"/>
</bean>
<bean id="mathCalculator" class="com.pe.beans.MathCalculator">
	<property name="complexNumber" ref="complexNumber"/>
</bean>
```

The above configuration works without any issue, but instead of configuring the complexNumber as a bean and injecting via property reference, we want to configure the complexNumber as a String literal value as 24, 34 where 24 is the base and 34 is the expo as shown below.

```
<bean id="mathCalculator" class="com.pe.beans.MathCalculator">
	<property name="complexNumber" value="24,34"/>
</bean>
```

In order to achieve the above, we need to write a custom property editor which will reads the string value "23, 34" in the configuration and coverts it to the target class attribute type, in this case ComplexNumber type and injects it.

So, to develop a custom property editor you need to write a class which extends from PropertyEditorSupport and should register this with the BeanFactory as shown below.

**ComplexNumberEditor.java**

```
package com.pe.beans;

import java.beans.PropertyEditorSupport;

public class ComplexNumberEditor extends PropertyEditorSupport {

	@Override
	public void setAsText(String value) throws IllegalArgumentException {
		int base = 0;
		int expo = 0;

		base = Integer.parseInt(value.substring(0, value.indexOf(",")));
		expo = Integer.parseInt(value.substring(value.indexOf(",") + 1,
				value.length()));
		ComplexNumber complexNumber = new ComplexNumber(base, expo);
		setValue(complexNumber);
	}
}
```

After developing a custom property editor class, you need to register this with the BeanFactory after creating the factory shown below.

**PETest.java**

```
package com.pe.test;

import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.pe.beans.ComplexNumber;
import com.pe.beans.ComplexNumberEditor;
import com.pe.beans.MathCalculator;

public class PETest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/pe/common/application-context.xml"));

		((ConfigurableListableBeanFactory) factory)
				.addPropertyEditorRegistrar(new PropertyEditorRegistrar() {

					@Override
					public void registerCustomEditors(
							PropertyEditorRegistry registry) {

	registry.registerCustomEditor(ComplexNumber.class,
									new ComplexNumberEditor());
					}
				});
		MathCalculator mc = factory.getBean("mathCalculator",
MathCalculator.class);
		mc.calculate();
	}

}
```

## 7.12 Internationalization

Spring supports the internationalization through ApplicationContext. Spring can load multiple properties files based on the base name and allow you to retrieve the messages from the properties files based on the Locale. In traditional applications if you want to read messages from resource bundle based on Locale, you need to load one bundle per one Locale.

Instead in spring it allows you to specify the Locale while fetching the message, so that it dynamically loads the bundle based on Locale and gets the message.

In order to support the same, spring has provided methods to access the messages from the resource bundle in ApplicationContext. In the ApplicationContext, it declared an attribute messageSource of type ResourceBundleMessageSource as shown below.

**Psuedo code**

```
public class ApplicationContext {
	@Autowire
	private ResourceBundleMessageSource messageSource;
	public void setMessageSource(ResourceBundleMessageSource messageSource) {
		this.messageSource = messageSource;
	}
}
```

So in order to load properties file you need to declare a bean whose type is ResourceBundleMessageSource, it will takes an attribute baseName, for which you need to pass the name of the properties file. This bean has to be named with messageSource so that spring will automatically injects into ApplicationContext, there by which you can access the messages using the convenient methods provided in ApplicationContext like getMessage(String, Object[], Locale) etc.

Note: - You have to place the properties file under classes' folder (place it in src, will automatically shipped as part of classes).

**application-context.xml**

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
	<property name="basename" value="label"/>
</bean>
```

**I18NTest.java**

```
package com.i18n.test;

import java.util.Locale;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class I18NTest {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/i18n/common/application-context.xml");
		System.out.println("Message : "
				+ context.getMessage("empid.label", new Object[] {"is not",
"valid"}, Locale.getDefault()));
	}
}
```

## 7.13 Bean Post Processor

In spring, we use Bean Lifecycle to perform initialization on a bean. But using init-method, destroy-method or InitializingBean, DisposableBean we do initialization or destruction process for a specific bean which implement these.

If we have a common initialization process which has to be applied across all the beans in the configuration, Bean lifecycle cannot handle this. We need to use BeanPostProcessor.

BeanPostProcessor has two methods postProcessBeforeInitialization and postProcessAfterInitialization, these methods will be invoked for all the beans on the core container. postProcessBeforeInitialization method will be called after the core container has created the bean and before it performs injection. postProcessAfterInitialization method will be called after the core container has perfomed injections. For both of these methods the bean will be passed as parameter along with beanName on which the methods are fired.

In these methods the developer can write the Initialization logic to initialize the beans. So, you need to write a class which implements from BeanPostProcessor interface and override the methods and provide the logic for initialization. The same has been depicted in the below example.

**EmployeeDelegate.java**

```
package com.bpp.beans;

public class EmployeeDelegate {
	private EmployeeVO employeeVO;
	private EmployeeDao employeeDao;

	public void insert() {
		employeeVO.setEmpID("E32542");
		employeeVO.setName("John");
		employeeVO.setSalary(353.34f);
		// employeeVO.setLastModifiedDate(lastModifiedDate)
		employeeDao.insert(employeeVO);
	}

	public void setEmployeeVO(EmployeeVO employeeVO) {
		this.employeeVO = employeeVO;
	}

	public void setEmployeeDao(EmployeeDao employeeDao) {
		this.employeeDao = employeeDao;
	}
}
```

**EmployeeDao.java**

```
package com.bpp.beans;

public class EmployeeDao {
	public void insert(EmployeeVO employeeVO) {
		System.out.println("inserting employee : "
				+ employeeVO.getLastModifiedDate());
	}
}
```

**BaseVO.java**

```
package com.bpp.beans;

import java.util.Date;

abstract public class BaseVO {
	private Date lastModifiedDate;

	public Date getLastModifiedDate() {
		return lastModifiedDate;
	}

	public void setLastModifiedDate(Date lastModifiedDate) {
		this.lastModifiedDate = lastModifiedDate;
	}

}
```

**EmployeeVO.java**

```
package com.bpp.beans;

public class EmployeeVO extends BaseVO {
	private String empID;
	private String name;
	private float salary;

	public String getEmpID() {
		return empID;
	}

	public void setEmpID(String empID) {
		this.empID = empID;
	}

	public String getName() {
		return name;
	}

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

	public float getSalary() {
		return salary;
	}

	public void setSalary(float salary) {
		this.salary = salary;
	}

}
```

**BaseVOBeanPostProcessor.java**

```
package com.bpp.beans;

import java.util.Date;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class BaseVOPostProcessor implements BeanPostProcessor {

	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName)
			throws BeansException {
		if (bean instanceof BaseVO) {
			((BaseVO) bean).setLastModifiedDate(new Date());
		}
		return bean;
	}

	@Override
	public Object postProcessBeforeInitialization(Object arg0, String arg1)
			throws BeansException {
		return arg0;
	}

}
```

**application-context.xml**

```
<bean id="employeeVO" class="com.bpp.beans.EmployeeVO" />
<bean id="employeeDao" class="com.bpp.beans.EmployeeDao" />
<bean id="employeeDelegate" class="com.bpp.beans.EmployeeDelegate">
	<property name="employeeVO" ref="employeeVO" />
	<property name="employeeDao" ref="employeeDao" />
</bean>
<bean id="bpp" class="com.bpp.beans.BaseVOPostProcessor"/>
```

Once you create the BeanPostProcessor class, you need to register the postprocessor with the bean factory as shown below.

**BPPTest.java**

```
package com.bpp.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.bpp.beans.BaseVOPostProcessor;
import com.bpp.beans.EmployeeDelegate;

public class BPPTest {

	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/bpp/common/application-context.xml"));
		BaseVOPostProcessor bpp = factory.getBean("bpp",
				BaseVOPostProcessor.class);

		((ConfigurableListableBeanFactory)
factory).addBeanPostProcessor(bpp);
		EmployeeDelegate ed = factory.getBean("employeeDelegate",
EmployeeDelegate.class);
		ed.insert();
	}

}
```

So, in the above code after registering the bean postprocessor, when you try to fetch the EmployeeDelegate bean from the core container, while creating the delegate it will try to create EmployeeVO and EmployeeDao to inject into it.

So after the container creates these before performing injection it will call the postProcessBeforeInitialization and after performing the injection, it will call postProcessAfterInitialization.

## 7.14 Bean Factory Post Processor

If we want to perform post initialization after creating the BeanFactory and before creating the beans, we need to use BeanFactoryPostProcessor. It's a one of the kind of extension hooks that spring has provided to the developer to perform customizations on the configuration that is loaded by the Factory.

Spring has provided built-in BeanFactoryPostProcessor's to perform post processing, based on the type of post processor you use, you will get the respective behavior in your application. One of it is PropertyPlaceHolderConfigurer.

PropertyPlaceHolderConfigurer is a post processor which will reads the messages from the properties file and replaces the ${} place holder's of a bean configuration after the BeanFactory has loaded it.

Let's try to understand this with an example.

**ConnectionManager.java**

```
package com.bfpp.beans;

public class ConnectionManager {
	private String url;
	private String userName;
	private String password;
	public String getUrl() {
		return url;
	}

	public void setUrl(String url) {
		this.url = url;
	}

	public String getUserName() {
		return userName;
	}

	public void setUserName(String userName) {
		this.userName = userName;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

}
```

**application-context.xml**

```
<bean id="connectionManager" class="com.bfpp.beans.ConnectionManager">
	<property name="url" value="${db.url}"/>
	<property name="userName" value="${db.un}"/>
	<property name="password" value="${db.pwd}"/>
</bean>
```

If you create a BeanFactory with the above configuration, the ConnectionManager bean will gets created with url, username and password with ${} token values. Instead of this, if we want to replace the ${} values with property file key values, we need to configure PropertyPlaceHolderConfigurer. This post processor after loading the configuration by BeanFactory and before the factory creates the ConnectionManager bean, will replaces the ${} values with property key values as shown below.

**db.properties**

```
db.url=jdbc:odbc:thin@1521:XE
db.un=weblogic
db.pwd=welcome1
```

Adding to the above configuration, you need to configure the post processor and register this with the BeanFactory.

```
<bean id="pphc"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
	<property name="location" value="classpath:db.properties"/>
</bean>
```

**BFPPTest.java**

```
package com.bfpp.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.bfpp.beans.ConnectionManager;

public class BFPPTest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/bfpp/common/application-context.xml"));
		BeanFactoryPostProcessor bfpp = factory.getBean("pphc",
BeanFactoryPostProcessor.class);
		bfpp.postProcessBeanFactory((ConfigurableListableBeanFactory)factory);

		ConnectionManager cm = factory.getBean("connectionManager",
ConnectionManager.class);
		System.out.println("cm.url : " + cm.getUrl());
	}

}
```

## 7.15 Event Processing

Spring provides Event handling capabilities using the ApplicationContext. For any event based processing technic we have four actors, 1) source 2) event 3) event listener and 4) event handler.

Source is the actor who raises an Event, Event is the class which contains the data representing the purpose of the event. Listener is the person who will listens for the event, and upon raising the event by source, listener will catches it and raises a method call on the handler to process that event.

So, Event handling is used for asynchronous processing, in spring an Event is represented with ApplicationEvent, Listener is created using ApplicationListener, and the application listener will contains the method onApplicationEvent() which will be fired upon raising the Event. In order to publish an event, the source needs an ApplicationEventPublisher. The below example shows the same.

**RefreshEvent.java**

```
package com.ep.beans;

import org.springframework.context.ApplicationEvent;

public class RefreshEvent extends ApplicationEvent {
	private String tableName;

	public RefreshEvent(Object source, String tableName) {
		super(source);
		this.tableName = tableName;
	}

	public String getTableName() {
		return tableName;
	}

}
```

**RefreshEventListener.java**

```
package com.ep.beans;

import org.springframework.context.ApplicationListener;

public class RefreshEventListener implements ApplicationListener<RefreshEvent> {

	@Override
	public void onApplicationEvent(RefreshEvent event) {
		System.out.println("Refreshing table : " + event.getTableName());
	}

}
```

**RefreshEventSource.java**

```
package com.ep.beans;

import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class RefreshEventSource implements ApplicationEventPublisherAware {
	private ApplicationEventPublisher publisher;

	public void raiseRefresh(String table) {
		publisher.publishEvent(new RefreshEvent(this, table));
	}

	@Override
	public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
		this.publisher = publisher;
	}

}
```

**application-context.xml**

```
<bean class="com.ep.beans.RefreshEventListener"/>
<bean id="refreshEventSource" class="com.ep.beans.RefreshEventSource"/>
```

**EPTest.java**

```
package com.ep.test;

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

import com.ep.beans.RefreshEventSource;

public class EPTest {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/ep/common/application-context.xml");

		RefreshEventSource res = context.getBean("refreshEventSource",
				RefreshEventSource.class);
		res.raiseRefresh("TBLEMPLOYEE");
	}

}
```

## 7.16 Bean Factory VS Application Context

| Bean Factory | Application Context |
|---|---|
| • Bean Factory is a lazy initializer, this means after creating the Bean Factory, it will loads the configuration file but will not create any beans for the bean declarations. When you try to fetch the bean by using factory.getBean(“”), Bean Factory will creates the bean.<br>• Bean Factory doesn’t support Internationalization.<br>• Bean Factory doesn’t supports Event Handling and Event processing<br>• Bean Factory will not automatically registers BeanPostProcessors or BeanFactoryPostProcessors, we need to explicitly write the code for registering them. | • ApplicationContext is an eager initializer. When you first create the application context with the configuration, after loading the configuration, it will immediately create all the beans in the configuration.<br>• Application Context supports internationalization.<br>• Application Context supports Event processing.<br>• Application Context up seeing the BeanPostProcessor and BeanFactoryPostProcessor declarations in the configuration, will automatically registers them with the container and applies processing. |

# Spring Annotation

# 8 Spring Annotation Support

## 8.1 Introduction to J2EE Annotation

Annotation is the code about the code that is metadata about the program itself. In other words information about the code is provided at the source code. Annotations are parsed/processed by compilers, annotation-processing tools and also executed at runtime.

Annotations have been introduced in JDK 1.5. It allows the programmers to specify the information about the code at the source code level itself. There are several ways we can use annotations. Few helps in understanding the source code (like documentation assistance) @override is the annotation that will be used when we override a method from base class. This annotation doesn't have any impact on run-time behavior, rather it helps javadoc complier to generate documentation based on it.

Apart from these annotations we have another way in using annotations which assist source code generators to generate source code using them. If we take example as Web Services, in order to expose a class as web service, we need to mark the class as @WebService. This annotation would be read by a tool and generates class to expose that class as web service.

Along with this there is another way, where when you mark a class with annotation, the run-time engine will executes the class based on that annotation with which it marked. For example, when you mark a class with @EJB, the class would be exposed as Enterprise Java Bean by the container rather than a simple pojo.

By the above we can understand that using annotations, a programmer can specify the various behavioral aspects of your code like documentation, code generation and run-time behavior. This helps in RAPID application development where in instead of specifying the information about your code in xml configuration file, the same can be specified by marking your classes with annotations.

Note: - Always your annotation configuration will be overwritten with the xml configuration if provided.

## 8.2 Spring Annotation Support

Spring support to annotations is an incremental development effort. Spring 2.0 has a very little support to annotation, when it comes to spring 2.5; it has extended its framework to support various aspects of spring development using annotations. In spring 3.0 it started supporting java config project annotations as well.

The journey to spring annotations has been started in spring 2.0; it has added @Configuration, @Repository and @Required annotations. Along with this it added support to declarative and annotation based aspectj AOP.

In spring 2.5 it has added few more annotations @Autowired, @Qualifier and @Scope. In addition in introduced stereotype annotations @Component, @Controller and @Service.

In spring 3.0 few more annotations has been added like @Lazy, @Bean, @DependsOn etc. In addition to spring based metadata annotation support, it has started adoption JSR - 250 Java Config project annotations like @PostConstruct, @PreDestory, @Resource, @Inject and @Named etc.

The below table list spring supported and Java Config project supported annotations.

| | | | |
|---|---|---|---|
| **Spring 2.0** | • @Configuration<br>• @Required<br>• @Repository | **Java Config Annotation Support** | • No Support |
| **Spring 2.5** | • @Autowired<br>• @Qualifier<br>• @Scope<br>Stereotyped annotations<br>• @Component<br>• @Service<br>• @Controller | | • No Support |
| **Spring 3.0** | • @Bean<br>• @DependsOn<br>• @Lazy<br>• @Value | | • @PostConstruct<br>• @PreDestroy<br>• @Resource<br>• @Inject<br>• @Named |

Let's explore these annotations with examples.

### 8.2.1 Working with @Configuration and @Bean

Instead of declaring a class as spring bean in a configuration file, you can declare it in a class as well. The class in which you want to provide the configuration about other beans, that class is called configuration class and you need to annotate with @Configuration. In this class you need to provide methods which are responsible for creating objects of your bean classes, these methods has to be annotated with @Bean.

Now while creating the core container, instead of using XMLBeanFactory, you need to create it using AnnotationConfigApplicationContext by passing the configuration class as input.

**AppConfig.java**

```
@Configuration
public class AppConfig {
  @Bean
  public MyService myService() {
     return new MyServiceImpl();
  }
}
```

**MyService.java**

```
public class MyService {
        public void doSomeStuff() {
                // some dummy logic
        }
}
```

**ConfigurationTest.java**

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
MyService myService = ctx.getBean(MyService.class);
myService.doSomeStuff();
```

### 8.2.2 Working with @Required

In spring 1.x onwards we have dependency check. For a bean if you want to detect unresolved dependencies of a bean, you can use dependency check. In this IOC container will check whether all the bean dependencies, which is expressed in its properties are satisfied or not. The problem with dependency check is you can't impose restriction on a certain property, either you need to check the dependencies on all simple or object or all.

In 2.0, it has introduced an annotation @Required and dependency check has been completely removed in 2.5. Using @Required annotation you can make, a particular property has been set with value or not.

**Engine.java**

```
package com.annotation.beans;

public class Engine {
	private Integer id;
	private String type;

	public Integer getId() {
		return id;
	}

	@Required
	public void setId(Integer id) {
		this.id = id;
	}

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}
}
```

You can use @Required on a setter level to mark it as mandatory. Simply using @Required annotation will not enforce the property checking, you also need to register an RequiredAnnotationBeanPostProcessor in the configuration file as shown below.

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
	<property name="id" value="22"/>
	<! - -if you don't provide value for id, raises error - ->
	<property name="type" value="T1"/>
	<qualifier value="myengine"/>
</bean>
(<context:annotation-config/>

(OR)

<bean class=
"org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"
/>)
```

### 8.2.3 Working with @Autowire

You can enable autowiring using @Autowire annotation rather than configuration approach. Unlike your declarative <bean autowire="byName/byType/constructor"/> etc, in annotation driven we don't have any modes.

In annotation based approach you may mark an attribute of a target class or a setter or a constructor or any arbitrary method for injecting the dependent object. In all the cases it will performs autowiring byType.

@Autowired annotation has an attribute required=true (@Autowired(required=true)). By default required is true. When you use required=true, while creating the target class object it will try to find the appropriate dependent class object in IOC container, if it couldn't find one the container will throws exception without creating core container.

At any point of time out of available number of constructors, you can mark only one constructor with @Autowired(required=true) and remaining should be set to false.

Below examples shows various ways of using @Autowired

**Attribute Level**

```
package com.annotation.beans;

import org.springframework.beans.factory.annotation.Autowired;

public class Motor {
	@Autowired(required=false)
	private Engine engine;

	public void run() {
		System.out.println("running with engine id : " + engine.getId());
	}
}
```

**Setter Level**

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

**Constructor Level**

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public Motor(Engine engine) {
                this.engine = engine;
        }
….
}
```

**Arbitrary method**

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void newEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

**application-context.xml**

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:annotation-config/>
```

In order to detect @Autowire either you need to use <context:annotation-config/> or need to declare a bean whose class is org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.

### 8.2.4 Working with @Qualifier

If you have more than one beans of type Engine in the configuration, as @Autowire will perform injection byType, it will not be able to make the decision of which bean has to be injected and will raise ambiguity error. To resolve this we need to perform byName, this can be done with @Qualifier as shown below.

**Using Qualifier**

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        @Qualifier("engine2")
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

**application-context.xml**

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<bean id=”maruthiengine” class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="engine2"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:annotation-config/>
```

Only the bean whose qualifier value is “engine2” will be injected into Motor class engine attribute, so indirectly using @Qualifier we are able to perform byName injection.

### 8.2.5 Working with stereotype annotations @Component, @Repository, @Service and @Controller

In addition to @Repository annotation in spring 2.0, spring 2.5 has added stereotype annotations @Component, @Service and @Controller to make your classes as spring beans.

When you mark you class with any of the above annotations those classes will be exposed as spring beans by the container. Based on the type of class you are trying to expose you need to use appropriate annotations.

@Component – This acts as a more generic stereotype annotation to manage any component by spring, whereas the other annotations are specialized for the specific use cases.

@Repository – This is used for exposing a DAO class as a spring bean. Even though database specific semantics are not imposed by using it, it helps you in applying exception translations on these @Repository classes using AOP.

@Service – The service layer class are annotated with @Service, even though the current release of the spring doesn’t has any impact on using it (other than exposing that class as spring bean), but future releases of spring might add some specializations on those classes.

@Controller – When you mark a class with @Controller, it will be exposed as spring MVC controller to handle form submissions.

In a spring core/spring jdbc applications you can use @Component, @Service or @Repository whereas @Controller can be used only in a Spring MVC application.

The below examples shows how to use @Component, but you can mark that class with @Service or @Repository as well, where the end effect would be same.

**Motor.java**

```
(ignored few sections for clarity)

@Component("motor")
public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

With the above configuration, the class will be exposed as spring bean with id "motor". In order to detect the @Component or stereotype annotations you need to use the tag <context:component-scan base-package="PKG NAME OF THE CLASSES"/> as shown below.

application-context.xml

```
<context:component-scan base-package="com.annotation.*"/>
```

You can retrieve the bean with context.getBean("motor").

## 8.3 Spring Java Config annotations

As explained earlier from spring 3.x it has added support to Java Config annotation support, this means when you mark you classes with any of the above discussed annotations like @Component or @Autowired, you classes will be tightly coupled with spring framework and will lose invasive feature of spring.

In order to retain loosely coupled feature, it spring has added support to java annotations, so when we use these annotations in spring applications, those will be read by spring container and add spring specific behavior to your classes. If you use in J2EE environment, those will behave based on container specification.

### 8.3.1 Working with @Inject

@Inject is a j2ee specific annotation, used for injecting/autowiring one class into another. This is more similar to @Autowired spring annotation. But the difference between them is @Autowire supports required attribute where @Inject doesn't has it.

@Inject also injects a bean into another bean byType as similar to @Autowired. The advantage of @Inject is it is from java (javax.inject package) which means even you separate your application from spring, still your classes can work with java rather than bounded to spring.

**Motor.java**

```
(ignored few sections for clarity)

@Component("motor")
public class Motor {
        private Engine engine;

        @Inject
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

Along with this you need to use <context:component-scan base-package=""/> to detect it by core container.

### 8.3.2 Working with @Named

There are two usages of @Named annotation

1) When you try to inject a bean using @Inject annotation it will performs injection byType, if you have more than one beans of that type in the application, the container will throw ambiguity error. In order to resolve it you need to use @Named annotation.

**application-context.xml**

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<bean id=”maruthiengine” class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:component-scan base-package="com.annotation.*"/>
```

In the above configuration you have two beans of type Engine, so when you try to inject the Engine into Motor by using @Inject, it will not be able to detect which bean has to be injected. So you need to mark the Engine attribute with @Named along with @Inject indicating the bean you want to inject show below.

**Motor.java**

(ignored few sections for clarity)

```
@Component(“motor”)
public class Motor {
        private Engine engine;

        @Inject
        @Named(“maruthiengine”)
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

2) Another use of @Named is instead of using the stereotype annotations of spring to expose your classes as spring beans, you can mark your class with @Named to expose it as spring bean show below.

**Motor.java**

```
(ignored few sections for clarity)

@Named("motor")
public class Motor {
        private Engine engine;

        @Inject
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

### 8.3.3 Working with @Resource

Instead of using @Inject, you can also use @Resource, the difference between @Inject and @Resource it @Inject will perform the injection byType where as @Resource will perform the injection byName.

**Motor.java**

```
(ignored few sections for clarity)

@Named("motor")
public class Motor {
        private Engine engine;

        @Resource
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
….
}
```

So, we need to have a bean whose id is "engine" in the configuration or with the component declaration with this name.

**application-context.xml**

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<context:component-scan base-package="com.annotation.*"/>
```

### 8.3.4 Working with @PostConstruct and @PreDestroy

As we discussed, Bean Lifecycle you can perform Initialization on a bean after injecting the dependent objects using init-method declaration or InitializingBean interface afterPropertiesSet and Destruction on a bean while removing a bean from core container using destroy-method or DisposableBean interface destroy() method.

Along with the above two ways, you can mark any arbitrary method on a class with @PostContruct and @PreDestory annotations. When you mark a method with @PostConstruct annotation, this method will be invoked by core container after injecting the dependent objects into the bean. When you mark a method with @PreDestory annotation, this method will be invoked as part of bean destruction process.

So, the @PostContruct and @PreDestory or annotation based lifecycle methods.

**Motor.java**

```
@Named("motor")
public class Motor {
        private Engine engine;

        @PostConstruct
        public void init() {
                // I will be invoked after performing injection
        }

        @Resource
        public void setEngine(Engine engine) {
                this.engine = engine;
        }

        @PreDestory
        public void release() {
                // I will be invoked while I am removing from container
        }
….
}
```

# 9 Aspect Oriented Programming (AOP)

Aspect oriented programming (in short AOP) is a new programming paradigm or technic similar to Object oriented programming. AOP allows you to apply cross cutting concern across various components of your application.

Here the concern is a piece of code, when it is cross-cutting concern; it means the code which has to be applied across various components/classes of your application. In general cross-cutting concern is helper code which compliments your business logic, this indicates without cross-cutting concern your business logic can exists.

Generalized examples of cross-cutting concerns are Auditing, Security, Logging, Transactions, Profiling and Caching etc.

In a traditional OOPS application if you want to apply a piece of code across various classes, you need to copy paste the code or wrap the code in a method and call at various places. The problem with this approach is your primary business logic is mixed with the cross-cutting logic and at any point of time if you want to remove your cross-cutting concern; you need to modify the source code. So, in order to overcome this, AOP helps you in separating the business logic from cross-cutting concerns.

By the above, we can understand that AOP compliments OOP but never AOP replaces OOP. The key unit of modularity in OOP is class, whereas in AOP it is Aspect. As how we have various principles of OOP like abstraction, encapsulation etc., AOP also has principles describing the nature of its use. Following are the AOP principles.

## 9.1 AOP Principles

1) Aspect – Aspect is the piece of code that has to be applied across various classes of the application
2) JoinPoint – The point at which you want to apply the aspect logic, generally in spring you can apply an aspect at method execution
3) Advice – Action taken by an aspect at a particular JoinPoint. Advice indicates how you want to apply the aspect on a joinpoint. There are multiple types of advices like before advice, after returning advice, around advice and throws advice
4) Pointcut – Collection of joinpoint representing on whom you want to advice the aspect
5) Target – The class on which you want to advice the aspect.
6) Weaving – The process of advising a target class with an aspect based on a pointcut to build proxy
7) Proxy – The outcome of weaving is called proxy, where the end class generated out of weaving process contains cross-cutting logic as well.

AOP is not something specific to spring; rather it is a programming technic similar to OOP, so the above principles are generalized principles which can be applied to any framework, supporting AOP programming style.

There are many frameworks in the market which allows you to work with AOP programming few of them are Spring AOP, AspectJ, JAC (Java aspect components), JBossAOP etc. Among which the most popular once are AspectJ and Spring AOP.

Let's try to compare the features between Spring AOP and AspectJ AOP.

| **Spring AOP** | **AspectJ AOP** |
|---|---|
| • Only supported joinpoint is method execution<br>• Spring supports run-time weaving; this means the proxy objects will be built on fly at runtime in the memory.<br>• Spring supports static and dynamic pointcuts | • It supports various types of Joinpoints like constructor execution, method execution, field set or field get etc.<br>• AspectJ uses compile-time weaving; this indicates your proxy classes will be available whenever you compile your code.<br>• AspectJ supports only static pointcuts |

At the time spring released AOP, already aspectj AOP has acceptance in the market, and seems to be more powerful than spring AOP. Spring developers instead of comparing the strengths of each framework, they have provided integrations to AspectJ to take the advantage of it, so spring 2.x not only supports AOP it allows us to work with AspectJ AOP as well.

In spring 2.x it provided two ways of working with AspectJ integrations, 1) Declarative programming model 2) AspectJ Annotation model. With this we are left with three ways of working with spring AOP as follows.

1) Spring AOP API (alliance API) – Programmatic approach
2) Spring AspectJ declarative approach
3) AspectJ Annotation approach

Even spring has integrated with AspectJ, it supports only few features of AspectJ described as below.

1) Still the supported Joinpoint is only a method execution.
2) Spring doesn't rely on AspectJ weaving capabilities, and uses its own weaving module so the weaving will happens at run-time.
3) As AspectJ doesn't support dynamic pointcut's, integrating with spring doesn't make any difference.

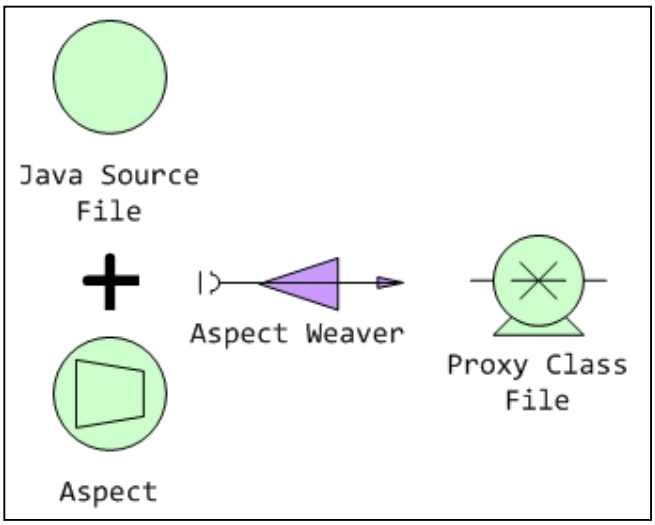



Diagram representing AOP Process

As described the weaving process will happen at run-time, in order to generate a class at run-time, spring uses proxy generation libraries. Spring supports two proxy generation libraries CGLib and JDKProxy run-time bytecode generation libraries to generate proxy classes on fly.

## 9.2 Types of Advices

Spring AOP implementation supports 4 types of advices, each one is described below.

1) Around advice – This is the advice that executes the aspect around the target class joinpoint, so here the advice method will be called before the target class method execution. So, the advice can control the target method execution.

2) Before advice – In this always the advice method executes before the target class method executes. Once the advice method finishes execution the control will not returned back to the advice method.

3) After Returning advice – Here the advice method executes after the target class method finishes execution, but before it returns the value to the caller.

4) Throws Advice – This advice method will be invoked only when the target class method throws an exception.

Let us try to understand how to work with various advices using different AOP programming styles in the following section.

## 9.3 Programmatic AOP

We use Spring AOP API's to work with programmatic AOP. As described programmatic aop supports all the types of advices described above. Let us try to work with each advice type in the following section.

### 9.3.1 Around Advice

Aspect is the cross-cutting concern which has to be applied on a target class; advice represents the action indicating when to execute the aspect on a target class. Based on the advice type we use, the aspect will gets attached to a target class. If it is an around advice, the aspect will be applied before and after, which means around the target class joinpoint execution.

If we have a usecase which demands for applying a cross-cutting concern around the target class joinpoint, we need to use around advice, one of the example for this caching.

Always the around advice method will be executed before your target class method executes by passing the entire target method call information to it. In an around advice we have three control points described below.

1) We have control over arguments; it indicates we can modify the arguments before executing the target class method.
2) We can control the target class method execution in the advice. This means we can even skip the target class method execution.
3) We can even modify the return value being returned by the target class method.

In order to work with any advice, first we need to have a target class, let us build the target class first.

**Math.java**

```
package com.aa.beans;

public class Math {
	public int add(int a, int b) {
		return a + b;
	}

	public int multiply(int a, int b) {
		return a * b;
	}
}
```

After creating the target class, we need to code the aspect representing the advice it is using. Create a class for example LoggingAspect which implements from MethodInterceptor and need to override the method invoke; this method accepts a parameter MethodInvocation, into which the entire information about the actual method would be passed.

You can access the original parameters with which the add method was called from MethodInterceptor by calling getArguments method on it. Along with this you can control the method execution by calling methodInterceptor.proceed(); Up on calling the proceed() method, the target method executes, if we don't call proceed() method, the target class method will never get executed. Up on finishing the target method execution the return value being returned by target method will be given to advice method. Here the advice can modify the return value and can return to the caller show below.

**LoggingAspect.java**

```
package com.aa.beans;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LoggingAspect implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation methodInvocation) throws Throwable {
		Object args[] = methodInvocation.getArguments();
		String methodName = methodInvocation.getMethod().getName();
		// log statement before execution
		System.out.println("entering into method : " + methodName + "("
				+ args[0] + "," + args[1] + ")");

		// modify arguments before calling
		args[0] = (Integer) args[0] + 10;
		args[1] = (Integer) args[1] + 10;
		// proceed() calls the target method add
		Object ret = methodInvocation.proceed();

		// log statement after exection
		System.out.println("exiting the method : " + methodName + "(" +
args[0]
				+ "," + args[1] + ") return value : " + ret);

		// modify return value
		ret = (Integer) ret + 10;
		return ret;
	}
}
```

In the above code the invoke method is executed when we call add method on the proxy object, by passing the information about the method call to MethodInvocation argument. Before calling the proceed() method, we are modifying the parameters, so that with these modified values proceed will be called. Once the target method execution finishes it returns the value as object in the proceed() method call. Here the advice can modify the return value and can return to the called.

Once the target and advice has been built, we can perform weaving to attach the aspect to the target. In order to perform weaving to build proxy, we need to use ProxyFactory class. To the ProxyFactory we need to add Advice and supply the target class on to which you want to apply that advice. The ProxyFactory will apply the advice on that given target and generates an in-memory proxy class and instantiates and returns that object. As we are working on Programmatic AOP, the weaving process will be done programmatically.

**AATest.java**

```
package com.aa.test;

import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

import com.aa.beans.LoggingAspect;
import com.aa.beans.LoggingDynamicPointcut;
import com.aa.beans.LoggingStaticPointCut;
import com.aa.beans.Math;

public class AATest {

	public static void main(String[] args) {
		ProxyFactory pf = new ProxyFactory();
		pf.addAdvice(new LoggingAspect()));
		pf.setTarget(new Math());

		Math math = (Math) pf.getProxy();
		System.out.println("Sum : " + math.add(4, 20));
		//System.out.println("Multiplication : " + math.multiply(1, 2));
	}
}
```

### 9.3.2 Before Advice

In a Before advice always the advice method executes before your target class joinpoint executes. Once the advice method finishes execution the control will be automatically transferred to the target class method and will not be returned back to the advice. So, in a Before Advice we have only one control point, which is only we can access arguments and can modify them. Below list describes the control points in a Before Advice.

1) Can access and modify the arguments of the original method.
2) Cannot control the target method execution, but we can abort the execution by throwing an exception in the advice method.
3) Capturing and modifying the return value is not applicable as the control doesn't return back to the advice method after finishing you target method.

In order to create a Before Advice, we need to first build a target class. Then we need to create an Aspect class which implements from MethodBeforeAdvice and should override before method. This method has three arguments java.reflect.Method method, Object[] args and Object targetObject. Using these paramters we can access the original method arguments and the method information.

Even we can modify the values in the args array so that with these modified values the target method will gets invoked. Common applications of Before Advice are Security, Authorization etc. Following examples shows the same.

**LoanCalculator.java**

```
package com.ba.beans;

public class LoanCalculator {
	public float calculateInterest(Long principle, int noOfYears,
			float rateOfInterest) {
		return (principle * noOfYears * rateOfInterest) / 100;
	}
}
```

LoanCalculator is the target class, before calling the calculateInterest, we want to check whether the caller is logged in and authenticated to call the method or not. We can check this in the calculateInterest() method itself, but if we don't want impose security check for this method, we need to modify source of this method to get rid of cross-cutting security code. Instead we will handle this security check to a MethodBeforeAdvice, where in the advice method we will check for security and if the security check has been passed then the control will be passed to calculateInterest() method otherwise the advice method will throw exception.

**UserInfo.java** (Class holding username and password values)

```
package com.ba.beans;

public class UserInfo {
	private String userName;
	private String password;

	public UserInfo(String userName, String password) {
		this.userName = userName;
		this.password = password;
	}

	// setter and getters on userName and password attributes
}
```

**AuthenticationManager.java** (helper class to perform login, authenticate and logout)

```
package com.ba.beans;

public class AuthenticationManager {
	private static ThreadLocal<UserInfo> threadLocal = new
ThreadLocal<UserInfo>();

	public void login(String un, String pwd) {
		threadLocal.set(new UserInfo(un, pwd));
	}

	public void logout() {
		threadLocal.set(null);
	}

	public boolean isAuthenticated() {
		boolean flag = false;
		UserInfo userInfo = threadLocal.get();
		if (userInfo != null) {
			if (userInfo.getUserName().equals("john")
					&& userInfo.getPassword().equals("welcome1")) {
				flag = true;
			}
		}
		return flag;
	}
}
```

**SecurityAspect.java** (Aspect class which executes before target method execution)

```
package com.ba.beans;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAspect implements MethodBeforeAdvice {

	@Override
	public void before(Method method, Object[] args, Object target)
			throws Throwable {
		System.out.println("entering into method : " + method);

		// check authenticated
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == false) {
			throw new IllegalAccessException("Invalid username/password");
		}

		// if authentication success
		// modify arguments
		args[0] = (Long) args[0] + 33;
	}
}
```

The code for building the proxy is same as for around advice. In the above the before method in the advice will executes before the calculateInterest() method in LoanCalculator class, here we are checking whether the user is authenticated to access the method or not. If not authenticated, will throw IllegalAccessException and aborting the calculateInterest() method execution.

**BATest.java**

```
package com.ba.test;

import org.springframework.aop.framework.ProxyFactory;

import com.ba.beans.AuthenticationManager;
import com.ba.beans.LoanCalculator;
import com.ba.beans.SecurityAspect;

public class BATest {

	public static void main(String[] args) {
		ProxyFactory pf = new ProxyFactory();
		pf.addAdvice(new SecurityAspect());
		pf.setTarget(new LoanCalculator());
		LoanCalculator proxy = (LoanCalculator) pf.getProxy();
		AuthenticationManager am = new AuthenticationManager();
		am.login("john", "welcome1");

		System.out.println("Interest : "
				+ proxy.calculateInterest(10L, 1, 12.0f));
	}
}
```

### 9.3.3 After Returning Advice

In this the advice method will be executed only after the target method finishes execution and before it returns the value to the caller. This indicates that the target method has almost returned the value, but just before returning the value it allows the advice method to see the return value but doesn't allows to modify it. So, in a After returning advice we have the below control points.

1) We can see parameters of the target method, even we modify there is no use, because by the time the advice method is called the target method finished execution, so there is no effect of changing the parameter values.
2) You cannot control the target method execution as the control will enters into advice method only after the target method completes.
3) You can see the return value being returned by the target method, but you cannot modify it, as the target method has almost returned the value. But you can stop/abort the return value by throwing exception in the advice method.

In order to create an After returning advice, after building the target class, you need to write the aspect class implementing the AfterReturningAdvice and should override the method afterReturning, the parameters to this method are Object returnValue (returned by actual method), java.reflect.Method method (original method), Object[] args (target method arguments), Object targetObject (with which the target method has been called).

If you observe in the below example, the return value of afterReturning method is void, which means you cannot modify and return the return value.

**KeyGenerator.java**

```
package com.ar.beans;

import java.util.Random;

public class KeyGenerator {
	public int generateKey(int size) {
		Random random = new Random(size);
		random.setSeed(5);
		int key = random.nextInt();

		return key;
	}
}
```

**WeakKeyCheckerAspect.java** (aspect checks whether the key computed by KeyGenerator class is strong or weak key)

```
package com.ar.beans;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class WeakKeyCheckerAspcect implements AfterReturningAdvice {

	@Override
	public void afterReturning(Object retVal, Method method, Object[] args,
			Object target) throws Throwable {
		if ((Integer) retVal <= 0) {
			throw new IllegalArgumentException("Weak Key Generated");
		}
	}

}
```

### 9.3.4 Throws Advice

Throws advice will be invoked only when the target class method throws exception. The idea behind having a throws advice is to have a centralized error handling mechanism or some kind of central processing whenever a class throws exception.

As said unlike other advices, throws advice will be called only when the target throws exception. The throws advice will not catch the exception rather it has a chance of seeing the exception and do further processing based on the exception, once the throws advice method finishes execution the control will flow through the normal exception handling hierarchy till a catch handler is available to catch it.

Following are the control points a throws advice has.

1) It can see the parameters that are passed to the original method
2) There is no point in controlling the target method execution, as it would be invoked when the target method rises as exception.
3) When a method throws exception, it cannot return a value, so there is nothing like seeing the return value or modifying the return value.

In order to work with throws advice, after building the target class, you need to write a class implementing the ThrowsAdvice interface. The ThrowsAdvice is a marker interface this means; it doesn't define any method in it. You need to write method to monitor the exception raised by the target class, the method signature should public and void with name afterThrowing, taking the argument as exception class type indicating which type of exception you are interested in monitoring (the word monitoring is used here because we are not catching the exception, but we are just seeing and propagating it to the top hierarchies).

When a target class throws exception spring IOC container will tries to find a method with name afterThrowing in the advice class, having the appropriate argument representing the type of exception class. We can have afterThrowing method with two signatures shown below.

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

In the above signature, Method, args and target is optional and only mandatory parameter is subclassOfThrowable. If a advice class contains afterThrowing method with both the signatures handling the same subclassOfThrowable, the max parameter method will be executed upon throwing the exception by target class.

Below example shows the same.

**Thrower.java**

```
package com.ta.beans;

public class Thrower {
	public int willThrow(int i) {
		if (i <= 0) {
			throw new IllegalArgumentException("Invalid parameter i");
		}
		return i + 10;
	}
}
```

**ExceptionLoggerAspect.java**

```
package com.ta.beans;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;

public class ExceptionLoggerAspect implements ThrowsAdvice {
	public void afterThrowing(IllegalArgumentException ie) {
		System.out.println("thrown : " + ie.getMessage());
	}

	public void afterThrowing(Method method, Object[] args, Object target,
			IllegalArgumentException ie) {
		System.out.println("Exception thrown by : " + method.getName() + "("
				+ args[0] + ") with exception message : " +
ie.getMessage());
	}
}
```

### 9.3.5 Pointcut

In all the above examples, we haven't specified any pointcut while advising the aspect on a target class; this means the advice will be applied on all the joinpoints of the target class. With this all the methods of the target class will be advised, so if you want to skip execution of the advice logic on a specific method of a target class, in the advice logic we can check for the method name on which the advice is being called and based on it we can execute the logic.

The problem with the above approach is even you don't want to execute the advice logic for some methods, still the call to those methods will delegate to advice, this has a performance impact.

In order to overcome this you need to attach a pointcut while performing the weaving, so that the proxy would be generated based on the pointcut specified and will do some optimizations in generating proxies.

With this if you call a method that is not specified in the pointcut, spring ioc container will makes a direct call to the method rather than calling the advice for it.

Spring AOP API supports two types of pointcuts as discussed earlier. Those are static and dynamic pointcuts. All the pointcut implementations are derived from org.springframework.aop.Pointcut interface. Spring has provided in-built implementation classes from this Interface. Few of them are described below.

Static Pointcut
1) StaticMethodMatcherPointcut
2) NameMatchMethodPointcut
3) JdkRegexpMethodPointcut

Dynamic Pointcut
1) DynamicMethodMatcherPointcut

### 9.3.6 Static Pointcut

In order to use a StaticMethodMatcherPointcut, we need to write a class extending from StaticMethodMatcherPointcut and needs to override the method matches. The matches method takes arguments as Class and Method as arguments.

Spring while performing the weaving process, it will determines whether to attach an advice on a method of a target class by calling matches method on the pointcut class we supplied, while calling the method it will passes the current class and method it is checking for, if the matches method returns true it will advise that method, otherwise will skip advicing.

**LoggingStaticPointcut.java**

```
package com.aa.beans;

import java.lang.reflect.Method;

import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class LoggingStaticPointCut extends StaticMethodMatcherPointcut {

	@Override
	public boolean matches(Method method, Class<?> targetClass) {
		if (Math.class == targetClass && method.getName().equals("multiply")) {
			return true;
		}
		return false;
	}
}
```

While performing the weaving, we need to supply the pointcut as input to the ProxyFactory as shown below.

**BATest.java** (Skipped logic for clarity)

```
		ProxyFactory pf = new ProxyFactory();
		pf.addAdvisor(new DefaultPointcutAdvisor(new LoggingDynamicPointcut(),
new LoggingAspect()));
		pf.setTarget(new Math());

		Math math = (Math) pf.getProxy();
```

### 9.3.7 Dynamic Pointcut

If you observe in static pointcut we have hardcoded the class and method names on whom we need to advice the aspect. So, the decision of advising the aspect on a target would be done at the time of weaving and would not be delayed till the method call. In case of Dynamic Pointcut, the decision of whether to attach an aspect on a target class would be made based on the parameters with which the target method has been called.

**LoggingDynamicPointcut.java**

```
package com.aa.beans;

import java.lang.reflect.Method;

import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class LoggingDynamicPointcut extends DynamicMethodMatcherPointcut {
	@Override
	public boolean matches(Method method, Class<?> targetClass, Object[] args) {
		if (Math.class == targetClass && method.getName().equals("add")
				&& (Integer) args[0] > 5) {
			return true;
		}
		return false;
	}
}
```

Table representing the control points for several advices

| **Advice Type/Control Points** | When Advice Method executes | Access to Target method parameters | Control of method execution | Access to Return Value |
|---|---|---|---|---|
| AroundAdvice | Around Target Method | Can see and Modify | ✓ | Can see and Modify the return value |
| MethodBeforeAdvice | Before Target Method executes | Can see and Modify | Don't have control and target method execution, only can abort target method exection | Not applicable (Control will not come back to advice method after finishing the target method execution) |
| AfterReturningAdvice | After Target method finished execution, but before returning value to the called | Can see the parameters | Not applicable | Can see the return value, but cannot modify them. Can abort returning the value to the called by throwing exception |
| ThrowsAdvice | Only when the target method throws exception | Can see paramters | Not applicable | Not applicable |

## 9.4 Declarative AOP

Spring 2.x has added support to declarative AspectJ AOP. The main problem with programmatic approach is your application code will tightly couple with spring, so that you cannot detach from spring. If you use declarative approach, your advice or aspect classes are still pojo's, you don't need to implement or extend from any spring specific interface or class. Your advice classes uses AspectJ AOP API classes, in order to declare the pojo as aspect you need to declare it in the configuration file rather than weaving it using programmatic approach.

The declarative approach also supports all the four types of advices and static pointcut. In the following section we will explore through the example how to work with various types of advices declaratively.

### 9.4.1 Around Advice

In this approach the aspect class is not required to implement from any spring specific class or interface, rather it should be declared as aspect in configuration file. In the aspect class you need to declare any arbitrary method with the following signature.

```
public Object methodName(ProceedingJoinPoint pjp)
```

This acts as an advice method. The return type of the method should be Object as the around advice has control over the return value. The advice method name could be anything but should take the parameter as ProceedingJoinPoint, the method on a target class is called JoinPoint in AOP, as we can control the execution of the Joinpoint in Around advice, so the parameter for this method should be ProceedingJoinPoint.

Using the ProceedingJoinPoint, you can access the actual method information similar to MethodInterceptor. You will declare this method as around advice method in the configuration file using <aop:config> tag, which is discussed in the example below.

In this example we are going to apply the aspect on the same Math.java class which we discussed in Programmatic approach.

**LoggingAspect.java**

```
package com.aa.beans;

import org.aspectj.lang.ProceedingJoinPoint;

public class LoggingAspect {
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		Object args[] = pjp.getArgs();

		System.out.println("entering into " + methodName + "(" + args[0] + ","
				+ args[1] + ")");
		// modify parameters
		args[0] = (Integer) args[0] + 10;
		Object ret = pjp.proceed(args);

		System.out.println("exiting from " + methodName + "(" + args[0] + ","
				+ args[1] + ") with return value (original) : " + ret);
		// modify ret val
		ret = (Integer) ret + 10;
		System.out.println("exiting from " + methodName + "(" + args[0] + ","
				+ args[1] + ") with return value (modified) : " + ret);
		return ret;
	}
}
```

If you see the above class, we haven't implemented or extended from any interface or class, so in order to make this as aspect class you need to provide this in the spring beans configuration file.

First we need to declare our target and aspect classes as beans. While declaring a class as a aspect, you need to import the "aop" namespace and need to use the tag <aop:config>. Inside the <aop:config> take we need to declare the bean as aspect using <aop:aspect ref="aspectbeanid">. Under this you need to declare which method you want to expose as advice method and how do you want to apply this method like around or before or afterReturning etc. Once this is done you need to supply the pointcut indicating on which target class methods you want to apply the aspect. The configuration has been show in the below code snippet.

**application-context.xml**

```
<bean id="math" class="com.aa.beans.Math"/>
<bean id="loggingAspect" class="com.aa.beans.LoggingAspect"/>

<aop:config>
	<aop:pointcut expression="execution(* com.aa.beans.Math.*(..))" id="pc1"/>
	<aop:aspect id="la1" ref="loggingAspect">
		<aop:around method="log" pointcut-ref="pc1"/>
	</aop:aspect>
</aop:config>
```

In the above declaration if you observe we have used a pointcut expression rather than referring to a class representing as pointcut. This pointcut expression is a static point cut expression, and it has been written in OGNL expression language. OGNL stands for Object Graph Navigation Language.

Expression starts with execution as a word representing apply the advice to all the class executions and the syntax is described below.

```
execution(<returntype>
<packagename>.<classname>.<methodname>(<arguments>))
```

When you create a core container with the following configuration, spring ioc container while creating will apply the advice on the target class (based on the pointcut) and creates proxy classes and instantiates these classes and host in IOC container.
In the above the "math" bean object will be created on the proxy after applying the advice, so when you request for context.getBean("math") instead of returing the original Math class object, IOC container will instantiates the proxy of Math class as the pointcut is matching to it and returns that object to you.

### 9.4.2 Before Advice

While working with Before Advice the entire approach is same like creating the aspect class and should declare a method, here the method should be public and the return type should be void as the Before advice cannot control the return value. The parameter to the advice method is JoinPoint rather than ProceedingJoinPoint as we don't have control on method execution.

Once we write a method in the aspect class following these rules, we need to declare that method as before advice method in the configuration shown below.

**SecurityAspect.java**

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;

public class SecurityAspect {
	public void check(JoinPoint jp, long principal, int noOfYears,
			float rateOfInterest) throws Throwable {
		System.out.println("Principal : " + principal);
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == Boolean.FALSE) {
			throw new IllegalAccessException("User not logged in");
		}

	}
}
```

application-context.xml

```
<aop:config>
	<aop:pointcut expression="execution(* com.ba.beans.*.*(..)) and
args(principal,noOfYears,rateOfInterest))"
		id="pc1" />
	<aop:aspect id="ap1" ref="securityAspect">
		<aop:before method="check" pointcut-ref="pc1" />
	</aop:aspect>
</aop:config>
```

In the above code if you observe the advice method along with JoinPoint, it took the parameters as principle, noOfYears and rateOfInterest as well, these are the parameters which are there on your target class method. If you want to access those parameters straight away in the advice method, you can declare your advice method to accept those values in appropriate arguments, and you need to specify those method parameters as arguments in pointcut expression as args(arg1, arg2, arg3). Here the arg1, arg2 and arg3 are the argument names of the advice method indicating the target method parameters has to be copied to these arguments respectively.

### 9.4.3 After Returning Advice

In the after returning advice while writing the advice method in the aspect, you need to have declared the method with return type as void as it cannot control the return value and the method takes two parameters. The first parameter is the JoinPoint and the second would be the variable in which you want to receive the return value of the target method execution.

**WeakKeyCheckerAspect.java**

```
public class WeakKeyCheckerAspect {
	public void checkKey(JoinPoint jp, int generatedKey) {
		// write the code to execute the logic with the return value.
	}
}
```

**application-context.xml**

```
<aop:config>
	<aop:pointcut expression="execution(* com.ar.beans.*.*(..))"
		id="pc1" />
	<aop:aspect id="ap1" ref="weakKeyCheckerAspect">
		<aop:afterReturning method="checkKey" returning=”generatedKey”
pointcut-ref="pc1" />
	</aop:aspect>
</aop:config>
```

### 9.4.4 Throws Advice

In this we need to write the aspect class with advice method taking the signature as below.

```
public void methodName([Method, args[], targetObject], subClassOfThrowable)
```

and the in declaration we need to declare the variable in which you are receiving the exception as throwing shown below.

**application-context.xml**

```
<aop:config>
	<aop:pointcut expression="execution(* com.ta.beans.*.*(..))"
		id="pc1" />
	<aop:aspect id="ap1" ref="loggingExceptionAspect">
		<aop:afterReturning method="handle" throwing=”ex” pointcut-ref="pc1"
/>
	</aop:aspect>
</aop:config>
```

## 9.5 AspectJ Annotation AOP

In this approach instead of using declarations to expose the classes as aspect and advice, we will annotate the classes with annotations. In order to make a class as aspect, we need to annotate the class with @Aspect. To expose a method as advice method, we need to annotate the method in aspect class with @Around or @Before or @AfterReturning or @AfterThrowing representing the type of advice. These annotations take the pointcut expression as value in them.

### 9.5.1 Working with advices

The same rules defined in declarative aop section applies in writing the advice method in annotation approach as well, but only difference is annotate the class as @Aspect and annotate the method with @Around("pointcut expression") shown below.

**LoggingSecurityAspect.java**

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class LoggingSecurityAspect {

	@Around("execution(* com.ba.beans.Math.*(..))")
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		System.out.println("entering into " + methodName);
		Object ret = pjp.proceed();
		System.out.println("exiting from " + methodName);
		return ret;
	}
}
```

In order to detect the annotations marked on the classes, we need to add <aop:aspect-autoproxy/> tag in the configuration.

If we are using the pointcut expression on the multiple advice methods, instead of re-writing the expression in all the places you can declare a method representing the pointcut expression. Annotate that method with @Pointcut("expression") and in the @Around or @Before or other advice annotations, use the method name as value representing the pointcut expression shows below.

**LoggingSecurityAspect.java**

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class LoggingSecurityAspect {
	@Pointcut("execution(* com.ba.beans.Math.*(..))")
	public void mathPointcut() {
	}
	@Around("mathPointcut()")
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		System.out.println("entering into " + methodName);
		Object ret = pjp.proceed();
		System.out.println("exiting from " + methodName);
		return ret;
	}
	@Before("mathPointcut()")
	public void check(JoinPoint jp) throws Throwable {
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == false) {
			throw new IllegalAccessException("Invalid user/pwd");
		}
		System.out.println("Authenticated succesfully");
	}
	@AfterThrowing(pointcut = "mathPointcut()", throwing = "ex")
	public void handle(IllegalAccessException ex) {
		System.out.println("I am in handle method");
		System.out.println("Exception Message : " + ex.getMessage());
	}
	@AfterReturning(pointcut = "mathPointcut()", returning = "sum")
	public void monitor(JoinPoint jp, int sum) {
		System.out.println("In monitor method");
		System.out.println("Returning value : " + sum);
	}

}
```

# Spring JDBC

# 10 Spring JDBC (Java Database connectivity)

Spring JDBC is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JPA in a consistent way. This allows us to switch between the above mentioned technologies fairly easy; along with it, one should not worry about how to handle exceptions in each technology.

Spring JDBC has provided a very strong exception hierarchy to handle any technology specific exception into spring jdbc exception or user-defined exception. It provides automatic translation of Technology specific exception into spring exceptions like for example a SQLException is translated into its own exception DataAccessException which is the root exception.

In a traditional JDBC, user has to write try-catch-finally blocks to handle the checked SQLException's. When it comes to spring JDBC without writing this annoying boiler plate code try-catch-block, spring will takes care of catching and handling these, so your code is freed from most of the glue code and you just need to focus on developing your business/persistence logic.

The main advantage of going with spring framework JDBC is you can avoid lot of boiler plate code that you write in a traditional JDBC, and spring jdbc will provide most of the stuff you do for example, creating a connection, statement etc and managing the resources like closing the connection and statements etc. We can better explain the advantages of using spring jdbc in a tabular fashion describing what spring jdbc offers to you and what you need to do.

Table describing the actions that spring provides and what you need to do apart from those.

| Action/Operation | Spring JDBC | You |
|---|---|---|
| Declaring connection configuration | | ✓ |
| Opening the connection | ✓ | |
| Supply SQL Statement | | ✓ |
| Declare parameters and provide values (positional parameters ? or named parameters :paramName) | | ✓ |
| Prepare statement and execute it | ✓ | |
| Loop through the resultset values (if any) | ✓ | |
| Wrap or extract the resultset values | | ✓ |
| Handle and process any exceptions | ✓ | |
| Handle Transactions | ✓ | |
| Managing and closing connection, statement and resultset | ✓ | |

## 10.1 Choosing an approach for JDBC Data access

Spring JDBC provides multiple approaches in working with data access logic. It provides the following approaches to work with Data access logic; JdbcTemplate, NamedParameterJdbcTemplate, SimpleJdbcTemplate, SimpleJdbcInsert, Mapping Sql Operations as sub classes.

1) JdbcTemplate - is the classic Spring JDBC approach and the most popular. All the other class in Spring JDBC will extends from this.
2) NamedParamterJdbcTemplate – This is a wrapper on JDBCTemplate and allows you to work with Named Parameters instead of (?) place holders. If you use (?) in you SQL Query, it would be tough to understand and track the values for those, so NamedParamterJdbcTemplate allows you to declare :NamedPrameter in the sql and provides methods to replace those :params while executing the query.
3) SimpleJdbcTemplate – This combines the most frequently used operations of JdbcTemplate and NamedParameterJdbcTemplate.

4) SimpleJdbcInsert and SimpleJdbcCall – In this one it optimizes the amount of coding and only you need to provide the table on which you want to perform in Insert and need to provide Map of values representing the table columns. These classes rely on Database Table meta data in performing the operations.
5) Mapping SQL Operations as Sub classes (Using SqlQuery and SqlUpdate classes) – In this approach you will create reusable sub-classes by declaring the query and their parameter types once and compile it. And you can execute various operations on it by passing different values. This is similar to how you work with JDO.

Spring JDBC Framework classes has been distributed across four packages those are namely core, datasource, object and support. The JdbcTemplate has been declared in core package and all the datasource related classes has been declare in datasource package. Support and Object contains some util and helper classes to make your jdbc work.

## 10.2 Types of supported JDBC Operations

On a RDBMS, we can perform various types of operations, among which while working with JDBC, we will most work on Data Manipulation and Data Query Language statements rather the other two. Let us detail which type of operations we perform in the below section.

1) Data Definition Language (DDL) – DDL statements are used for creating new schema or tables in a schema, this indicates these statements are used for setting up the database and its tables required for storing the data of your application. When we use JDBC, we try to access the existing data in a database rather creating a new database from Java Language, so we don't need to experiment on DDL.
2) Data Manipulation Language (DML) – These are the statements that supports inserting/manipulating/removing the data from the existing database tables. Here we might insert a single record/a set of records (bulk operations) into the database, spring jdbc supports both the types of operations.
3) Data Query Language (DQL) – This is used for querying information from a database table. There are different types of query operations we can perform on a table as below.
    a. Aggregate operations (COUNT, AVG, SUM etc.)
    b. Select a column in a row
    c. Select a Row as Object
    d. Select list of values/objects
4) Data Control Language (DCL) – Data control language is used for granting or revoking the access to various database objects in a database. This would be generally done by a Database administrator, so it would not be appropriate to perform these operations through JDBC.

So based on above described db operations that can be done on JDBC we would try to perform these operations using JdbcTemplate, NamedParameterJdbcTemplate and with all other approaches as well in the following sections.

## 10.3 Setting up DataSource

In order to perform anything in a JDBC application, we need a connection object. In a traditional core java jdbc applications, we create the connection to a database using DriverManager, even though every application needs a connection object to work with a database, developer has to write this logic in every application (this kind of code is called boiler plate code). Instead of repeatedly writing it, spring has provided a declarative configuration for creating a connection. In this we declare the parameters that are required to create a connection object and spring would be able to automatically provide the connection to our code.

In order to get a connection, we need to configure DataSource in the configuration. A DataSource is a class which contains configuration information using which it creates connection object. A class which implements javax.sql.DataSource Interface can be configured as a Datasource spring bean. Spring provides one of the implementations of java.sql.DataSource interface as part of its core package which is DriverManagerDataSource. This class is similar to DriverManager class in a traditional JDBC application. For this class we need to set the properties like DriverClass, Url, Usename and Password for creating a connection. The sample configuration has been show in the below fragment.

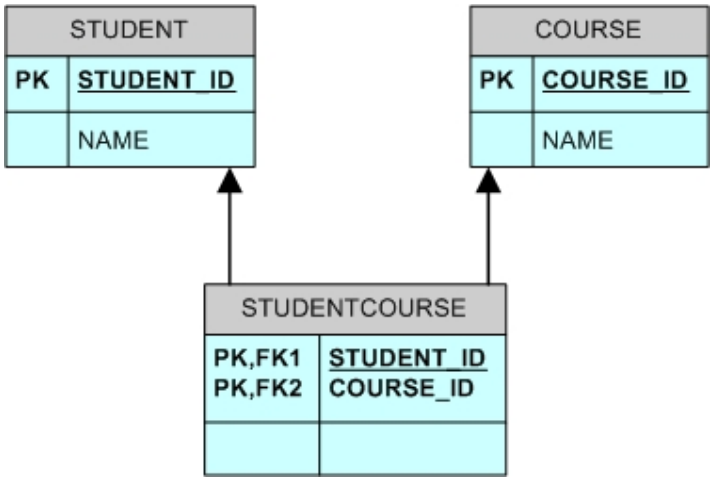
**persistence-beans.xml**

```
<bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="jdbc:oracle:thin:@//localhost:1521/xe" />
        <property name="username" value="hr" />
        <property name="password" value="hr" />
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
</bean>
```

The above dataSource bean is being injected into rest of the Spring Jdbc classes like JdbcTemplate or NamedParamterJdbcTemplate to execute Sql Operations.

## 10.4 Sample Schema and Table Structure

For explaining the examples on Spring Jdbc, we need some tables to execute operations, you might choose to work on these examples either on an Oracle database or MsSqlServer database, based on the database you are working you need to import the relevant Driver jar. Along with that you need to create some tables to understand and execute the examples which we are going to discuss, so here an E-R representation of the table structure has been given below based on that you should create your tables.




## 10.5 Using JDBCTemplate

JDBCTemplate is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors like forgetting to close a resource etc. Along with this it supports the basic jdbc workflow like creating a statement, executing a statement and iterating over the resultset etc. Here you need to write the business logic for using the resultset values. In this way the core functionality required to execute your sql query will be facilitated by JdbcTemplate.

Apart from this if you want to work with your own logic in executing the statements and iterating the resultset values etc, spring JdbcTemplate allows you to work with PreparedStatements and CallableStatements by providing classes like PreparedStatementCreator, PreparedStatementCallback and CallableStatementCreator, CallableStatementCallback etc. When working with those classes, these classes has two phases of execution 1) Prepare or create phase 2) Execute and Callback to handle the result phase. In the prepare phase you need to create the prepared statement by using the connection object passed to you by JdbcTemplate and in the CallBack phase you need to execute the statement and need to iterator over the resultset values. Here it seems like you are not getting most out spring jdbc, but the creation and closing of the resources has been taken care by JdbcTemplate itself.

By this we understood that working with JdbcTemplate involves two ways, one is straight away using the JdbcTemplate methods to get the resultset values and another is creating your own statements and call back handlers. We will discuss further these approaches using example.

### 10.5.1 Working with PreparedStatements using Jdbc Template

Spring JdbcTemplate provides lot of methods which supports executing an SQL statement and returning the results in pojo object. If you want to write your own logic of handling the resultset values rather than wrapping into a pojo object, then you need to create a statement and iterator over the resultset to perform business logic. JdbcTemplate exposes convenient methods which allow you to execute a PreparedStatement.

Executing a PreparedStatement involves two phases, creating or preparing the PreparedStatement using the PreparedStatementCreator class and executing and wrapping the resultset values in callback phase using PreparedStatementCallback.

Before creating the PreparedStatementCreator or Callback class to execute the statement, we need to setup the JdbcTemplate with a Datasource object. In order to execute any Sql Query we needs connection object and this would be provided by the DataSource bean which we discussed in the previous section. So, while JdbcTemplate performs any operation it will try to get the connection from the datasource with which we created it. The below code fragment shows how to configure a JdbcTemplate to take DataSource as dependent object.

**persistence-beans.xml**

```
<bean id="jdbcTempalate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dataSource" />
</bean>
```

Once you create a JdbcTemplate bean with the above configuration, you need to inject this bean into the Dao classes to perform Sql Operations.

**EmployeeDao.java**

```
package com.emp.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCallback;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.RowMapper;

import com.emp.business.Employee;

public class EmployeeDao {
	private JdbcTemplate jdbcTemplate;

	// insert datasource and create jdbcTemplate
	public EmployeeDao(DataSource dataSource) {
			jdbcTemplate = new JdbcTemplate(dataSource);
	}


	public List<Employee> getEmployeesByName(final String empName) {
		PreparedStatementCreator creator = new PreparedStatementCreator() {

			public PreparedStatement createPreparedStatement(
					Connection connection) throws SQLException {
				PreparedStatement ps = connection
						.prepareStatement("select * from tblemp where
emp_name like ?");
				ps.setString(1, empName);
				return ps;
			}
		};
```

In contd…

```
		PreparedStatementCallback<List<Employee>> callBack = new
PreparedStatementCallback<List<Employee>>() {

			public List<Employee> doInPreparedStatement(
					PreparedStatement preparedStatement) throws
SQLException,
					DataAccessException {
				List<Employee> employees = new ArrayList<Employee>();
				ResultSet rs = preparedStatement.executeQuery();
				while (rs.next()) {
					Employee e = new Employee(rs.getInt("emp_id"),
							rs.getString("emp_name"),
rs.getFloat("salary"));
					employees.add(e);
				}

				return employees;
			}
		};
		return jdbcTemplate.execute(creator, callBack);
	}
}
```

**persistence-beans.xml**

```
<!—either you can inject jdbcTemplate bean or inject dataSource and create
JdbcTemplate object by passing dataSource to its constructor →

<bean id="employeeDao" class=" com.emp.dao.EmployeeDao">
	<constructor-arg ref="dataSource" />
</bean>
```

### 10.5.1 Using JdbcTemplate Operations

As we discussed earlier let us try to explore how to perform various DML and DQL operations using the methods of JdbcTemplate class in the following sections.

#### *10.5.1.1 Aggregate Operations*

The aggregate operations that could be performed on a database are SUM, AVG, COUNT etc. All these functions will result in single columned results. So in order to execute an aggregate query using JdbcTemplate class, there is a method queryForInt(), to this method you need to pass the SQL where the result of execution of that query should yield an integer single columned value. Let us try to understand with an example how to work with it.

**StudentDao.java**

```
package com.jdbctemplate.dao;

import org.springframework.jdbc.core.JdbcTemplate;
import com.jdbctemplate.business.StudentBO;

public class StudentDao {
	private final String SQL_COUNT_OF_STUDENT = "SELECT COUNT(*) FROM
STUDENT";
	private JdbcTemplate jdbcTemplate;
	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}
	public int getCountOfStudents() {
		return jdbcTemplate.queryForInt(SQL_COUNT_OF_STUDENT);
	}
}
```

In the above code we called the method queryForInt to execute our sql query and give the result of execution as Integer rather than ResultSet. In further detail to it, we haven't created the Statement or ResultSet objects or even the connection rather we just configure the JdbcTemplate class as a bean in the configuration and inject into StudentDao as contructor argument, and JdbcTemplate is going to barrow the connection object and going to execute the sql query you passed to it.

Note: - In the above code the SQL Query has been declared as an constant String variable SQL_COUNT_OF_STUDENT. It is recommended to declare your queries as constants and pass those variables to your methods where you want to execute that SQL. This is not a Spring JDBC constraint rather this is J2ee Best Pratice so that you application code maintenance would be easy.

#### *10.5.1.2 Query Single column value*

You may query one value from a column for example finding a Student Name by Student Id; here the Student Id is a primary key so that always the result of query execution against a primary key column would yield only one value. As we are trying to find student name by primary key student id the result is one student name.

In order to query a single column value as Object, we need to use queryForObject() method. It has lot of variants, out of which one of it takes parameter as the SQL query and the type of value returned out of execution and Object array, as we are selecting only NAME by STUDENT_ID, the result of execution is a String value. In the WHERE clause of the query we need to specify the condition on STUDENT_ID =?. In this (?) is the place holder or substitution positional parameter for which we are going to provide the value while executing the query, which is show below.

**StudentDao.java**

```
package com.jdbctemplate.dao;

import org.springframework.jdbc.core.JdbcTemplate;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_FIND_NAME_BY_ID = "SELECT NAME FROM STUDENT
WHERE STUDENT_ID = ?";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public String getStudentNameById(int studentID) {
		return jdbcTemplate.queryForObject(SQL_FIND_NAME_BY_ID,
String.class,
				new Object[] { studentID });
	}
}
```

In the above example the query contains a (?) which is a place holder going to be replaced while executing it. If you observe while calling queryForObject method the first parameter is SQL query, second one is the type of outcome we are expecting after executing the query and third argument is Object array containing studentID value, this value will be replaced with the (?) in the query while executing. Let's say if your query contains two (?)'s, then you need to pass two values as part of your Object array so that the first (?) will be replaced with first value of the Object array and the second one with second value of the array respectively.

#### *10.5.1.3 Query Single Row as Object*

In order to query a record from a table, we need to use queryForObject() method, as we said there are multiple signatures of the queryForObject() method, the one we use here take parameters as queryForObject(SQL,Object[],RowMapper); If you observe the method signature, the third argument to the method is RowMapper.

The JdbcTemplate can execute the query which you have provided to it by replacing the (?) with Object[] values. But after executing the query, it cannot wrap the result values into your pojo. So the RowMapper is an Interface which contains a method mapRow(ResultSet, Row Index), you need to pass the object of RowMapper implementation class to the JdbcTemplate's, queryForObject method.

Once the method has finished executing the query it will passes the result set object to the mapRow method of the RowMapper object to convert the resultset values into Object.

So, you need to write the code for mapping the resultset values to object in the mapRow method and should provide it to JdbcTemplate, using which the JdbcTemplate will executes the query and build the object for you. This is shown in the below example.

**StudentDao.java**

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_FIND_STUDENT_BY_ID = "SELECT STUDENT_ID, NAME
FROM STUDENT WHERE STUDENT_ID = ?";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public StudentBO findStudent(int id) {
		return jdbcTemplate.queryForObject(SQL_FIND_STUDENT_BY_ID, new
StudentRowMapper(), new Object[] { id });
	}
```

```
	private static final class StudentRowMapper implements
RowMapper<StudentBO> {
		@Override
		public StudentBO mapRow(ResultSet rs, int rowIndex)
				throws SQLException {
			return new StudentBO(rs.getInt("STUDENT_ID"), rs
					.getString("NAME"));
		}
	}

}
```

**StudentBO.java**

```
package com.jdbctemplate.business;

public class StudentBO {
	private int student_id;
	private String name;

	public StudentBO(int student_id, String name) {
		this.student_id = student_id;
		this.name = name;
	}

	// setters and getters on the attributes

	@Override
	public String toString() {
		return ("Student ID : " + this.getStudent_id() + " Name : " + this
				.getName());
	}
}
```

#### *10.5.1.4 Query Multiple Rows as List of Objects*

This is similar to working with querying one row as object but the only difference is instead of calling queryForObject(), you need to call the simple query() method. Same is shown below

**StudentDao.java**

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_SELECT_ALL_STUDENTS = "SELECT STUDENT_ID,
NAME FROM STUDENT";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public List<StudentBO> getAllStudents() {
		return jdbcTemplate.query(SQL_SELECT_ALL_STUDENTS, new
StudentRowMapper());
	}
```

```
	private static final class StudentRowMapper implements
RowMapper<StudentBO> {
		@Override
		public StudentBO mapRow(ResultSet rs, int rowIndex)
				throws SQLException {
			return new StudentBO(rs.getInt("STUDENT_ID"), rs
					.getString("NAME"));
		}
	}

}
```

#### *10.5.1.5 Insert/Update/Deleting a record*

In order to perform Insert or Update or Delete operations on a table, we need to use the update method provided in JdbcTemplate class. This method will perform the Insert/Update/Delete based on the query you passed to it. The result of executing this method is an Integer indicating the number of rows affected due to the execution.

The update method takes arguments as update(SQL, Object[]), where the values in the Object[] would be replaced with the (?) place holders.

**StudentDao.java**

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDao {

	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public int insert(int student_id, String name) {
		return jdbcTemplate.update(SQL_INSERT_STUDENT, new Object[] {
				student_id, name });
	}
}
```

In the above example instead of passing INSERT query to the update method if you pass UPDATE SQL query it will performs update operation the same applies to DELETE as well.

#### *10.5.1.6 Batch Operations*

If we want to perform bulk insert or update operations on a database, we need to use batchUpdate method, this method takes parameter as BatchPreparedStatementSetter Interface implementation object.

The BatchPreparedStatementSetter interface has methods setValues() and getBatchSize(). The getBatchSize() method should return a Integer value indicating number of records you want to insert as part of operation. The setValues() method takes argument as PreparedStatement, the batchUpdate method while inserting the records it would call the setValues() method for BatchSize no of times by passing PreparedStatement object as parameter to you, in this method you need to set the values which you want to insert as part of this batch. The same is shown below.

**StudentDao.java**

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public int[] insert(final List<StudentBO> students) {
		return jdbcTemplate.batchUpdate(SQL_INSERT_STUDENT, new
StudentBatchSetter());
	}
```

```
	private static final class StudentBatchSetter implements
BatchPreparedStatementSetter {
		@Override
		public void setValues(PreparedStatement preparedStatement,
				int index) throws SQLException {
			preparedStatement.setInt(1, students.get(index)
					.getStudent_id());
			preparedStatement.setString(2, students.get(index)
					.getName());
		}
		@Override
		public int getBatchSize() {
			return students.size();
		}
	}
}
```

## 10.6 Using NamedParameterJDBCTemplate

While working with JdbcTemplate, if we want to pass any values to the queries while executing, we need to declare the (?) place holders in the query and need to pass the values for these place holders in an Object array. It would be tough to correlate the values you are passing the object array to a (?) field, as we completely dependent on order of passing, this approach is easily error prone and future changes involves lot of effort.

In order to overcome the above dis-advantage spring has introduces a class NamedParamterJdbcTemplate, it a wrapper on JdbcTemplate class, which means it exactly contains similar set of features that JdbcTemplate. In addition, NamedParameterJdbcTemplate allows you to declare named parameters in the query instead of (?) as ":varName".

So, while executing the query you can replace the ":varName" with the value. So that you will be sure that for which named parameter what is the value being passed and you don't need to rely on order of passing.

Basically you need to pass values for Named Parameters declare in a query. One way of passing the values is Map<String,Object> where string represents the named Paramter name and Object represents the value you want to pass for it.

Apart from this you can create an object of SqlParameterSource implementation class object; one of the implementation is MapSqlParameterSource, which represents data in terms of Map of Key and values. Below example shows how to pass values for named parameters declared in the query.

Another implementation of SqlParameterSource is BeanPropertyParamterSource this takes input as Object and creates Map of Key and value where key is the Object attribute name and value as its value.

The below example depicts all these scenarios.

**StudentDao.java**

```
package com.npjt.dao;

import com.npjt.business.StudentBO;

public class StudentDao {
	private final String SQL_SEARCH_STUDENT_BY_NAME = "SELECT STUDENT_ID,
NAME FROM STUDENT WHERE NAME LIKE :STUDENT_NAME";
	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(:studentId, :studentName)";

	private NamedParameterJdbcTemplate npJdbcTemplate;

	public StudentDao(NamedParameterJdbcTemplate npJdbcTemplate) {
		this.npJdbcTemplate = npJdbcTemplate;
	}

	public List<StudentBO> searchStudentByName(String name) {
		SqlParameterSource paramMap = new
			MapSqlParameterSource("STUDENT_NAME", name);

		return npJdbcTemplate.query(SQL_SEARCH_STUDENT_BY_NAME,
				paramMap, new StudentRowMapper());
	}

	public int insert(StudentBO student) {
		SqlParameterSource paramSource = new
				BeanPropertySqlParameterSource(student);
		return npJdbcTemplate.update(SQL_INSERT_STUDENT, paramSource);
	}

	private static final class StudentRowMapper implements
						RowMapper<StudentBO> {
		@Override
		public StudentBO mapRow(ResultSet resultSet, int rowNum)
				throws SQLException {
			return new StudentBO(resultSet.getInt("STUDENT_ID"),
					resultSet.getString("NAME"));
		}
	}
}
```

## 10.7 Mapping SQL Operations as Sub classes

As mentioned earlier there are multiple ways of working with spring jdbc, one of which is mapping the Sql operations as sub classes. This is similar to the concept of JDO where you define the query, declare the parameters and compile the query. Once you do that you can execute the query with various parameters.

In order to map sql operations as sub classes, spring jdbc has provided SqlQuery, SqlUpdate classes from which we need to extend our class. We need to provide the query and datasource as parameters while creating the sub class and need to compile the query.

### 10.7.1 Using SqlQuery

If you want to perform Select operations, you need to create a sub-class inside your DAO class which extends from SqlQuery class, but SqlQuery class is the base class, instead of using it, there is a another sub-class of SqlQuery which is MappingSqlQuery which has sophisticated method mapRow where you map a resultset values to an Object.

Once you create a sub-class from MappingSqlQuery, you need to pass the dataSource and Sql to be used as part of this class. Once you set up datasource and Sql Query for execution, you need to call super.compile() which will compiles the query only once after creating the object of it. Then there are two types of methods on the sub class available 1) executor methods and 2) finder methods. The executor method returns List of Objects and Finder methods returns only one object. Based on the nature of operation you perform you need to use appropriate methods to query the data.

**StudentDao.java**

```
package com.mso.dao;

public class StudentDao {
	private DataSource dataSource;

	public MapStudentDao(DataSource dataSource) {
		this.dataSource = dataSource;
	}

	public List<StudentBO> searchStudentsByName(String name) {
		return new SelectStudent(dataSource).searchStudentByName(name);
	}
```

Inner class in contd….

```
        private final class SelectStudent extends MappingSqlQuery<StudentBO> {

                public SelectStudent(DataSource ds) {
                        super(ds, "SELECT STUDENT_ID, NAME FROM STUDENT WHERE
                                                                        NAME = ?");
                        super.declareParameter(new SqlParameter("NAME",
                                                                Types.VARCHAR));
                        super.compile();
                }

                @Override
                protected StudentBO mapRow(ResultSet resultSet, int rowNum)
                                throws SQLException {
                        return new StudentBO(resultSet.getInt("STUDENT_ID"),
                                        resultSet.getString("Name"));
                }

                public List<StudentBO> searchStudentByName(String name) {
                        return execute("%N%");
                }
        }
}
```

### 10.7.1 Using SqlUpdate

SqlUpdate class encapsulates a Sql Upate. Like similar to SqlQuery, the SqlUpdate is also a reusable class. You can call the update() operations on this class with various parameters once the query is compiled. Unlike SqlQuery, this is a concrete class but if you want to provide customized update, you can create a inner class which extends from SqlUpdate and can execute the operations on it.

**StudentDao.java**

```
package com.mso.dao;

public class StudentDao {
        private DataSource dataSource;

        public MapStudentDao(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public int update(int studentId, String name) {
                return new UpdateStudent(studentId, name);
        }
```

Inner class contd….

```
        private final class UpdateStudent extends SqlUpdate {

                public UpdateStudent(DataSource ds, String sql) {
                        super(ds, "UPDATE STUDENT SET NAME=? WHERE STUDENT_ID =
                                                                                ?");
                        declareParameter(new SqlParameter(Types.VARCHAR));
                        declareParameter(new SqlParameter(Types.INTEGER));
                        compile();
                }

                public int update(int id, String name) {
                        return update(new Object[] { name, id });
                }
        }
}
```

## 10.8 SimpleJdbcInsert

SimpleJdbcInsert allows you to working with insert operations with minimal or no configuration. Here you don't need to provide the SQL query, rather you will pass the table name and the data map containing key as column name and value as Object type.

When you call execute method by passing the data map of values, it would automatically query the metadata of the table into his it has to perform insert operation and builds a query and inserts the data.

**StudentDao.java**

```
public class SimpleStudentInsertDao {
	private SimpleJdbcInsert simpleJdbcInsert;

	public SimpleStudentInsertDao(DataSource dataSource) {
		this.simpleJdbcInsert = new SimpleJdbcInsert(dataSource);
	}

	public void insert(StudentBO student) {
		simpleJdbcInsert.setTableName("STUDENT");
		SqlParameterSource paramSource = new
BeanPropertySqlParameterSource(student);
		simpleJdbcInsert.execute(paramSource);
	}

	public void insert(int id, String name) {
		simpleJdbcInsert.setTableName("STUDENT");
		Map<String, Object> data = new HashMap<String, Object>();
		data.put("STUDENT_ID", student.getStudentId());
		data.put("NAME", student.getStudentName());
		simpleJdbcInsert.execute(data);
	}
}
```

# Spring Transaction

# 11 Spring Transaction Support

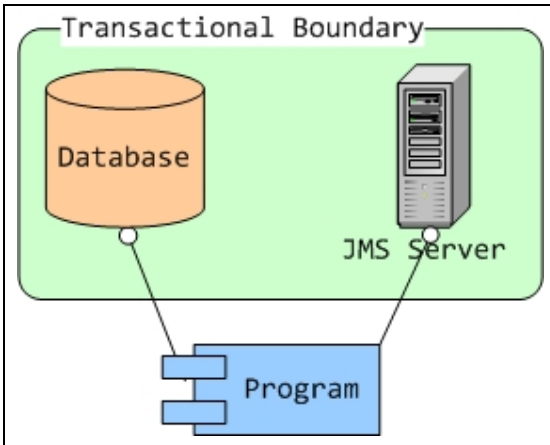
Spring transaction framework provides a consistent abstraction across transaction management API's. Any application extensively requires transaction management capabilities to handle its business operations consistently.

In a traditional Java or J2ee programming, developer has two choices for Transaction management. Those are Global or Local transactions; each of them has specific use cases where they have to be used.

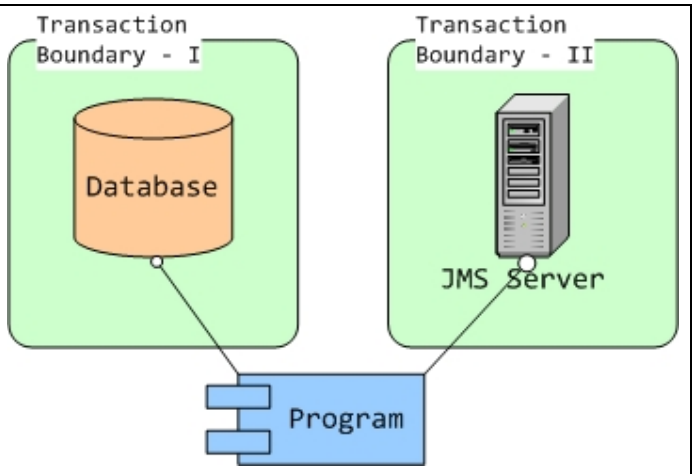
## 11.1 Global Transaction

Global transactions are also called as XA Transactions, a transaction is said to be Global or XA only when multiple resources are involved in it. Typically the resources would be a database and a message queues etc. When we say these two resources are involved in a transaction, it means performing the operation on a database and on a messaging queue should be part of single transaction and if a roll back happens it should roll back the operations on both the resources rather than one.

The below figure shows the transactional boundaries in a Global transaction and Local Transaction.




**Global (XA) transaction**




**Local Transaction**

The participating resources of a global transaction must be also XA resources rather than local resources. For example while configuring a Datasource to get the connection, the application server will give you an option of selecting the Datasource as XA or non-XA, this indicates whether the connections that are created out of that datasource will participate in global transactions or not.

### 11.1.1 Two-Phase commit

Global transaction will use the two phase commit process. This indicates the global transaction will be managed by a Transaction coordinator (also called as TransactionManager). Every resource in the transaction is maintained by the ResourceManager and executes the transaction as described below.

Phase – I
a) Each ResourceManager coordinates the local operations and writes them to log
b) If successful operations, response is OK

Phase – II -- If all ResourceManager's responds OK
a) Coordinator Issues commit on all resource managers
b) Participants complete the commit process and writes to log file.

Otherwise:
a) Coordinator issues roll back to all the resourcemanagers.
b) Participants undo's all the operations locally.

In order to work with Global Transactions or Distributed transactions, you need a Transaction Coordinator which will be maintained by a J2EE Server. To work with XA Transactions, J2EE has provided api's like JTA or EJB etc.

In JTA you need to programmatically manage transactions, and the API complex to work with and need to perform JNDI lookup to get the TransactionCordinator from J2EE server.

You can use declarative transaction management using EJB's, but EJB's are heavy wait components and may not be appropriate for whom who are looking for only Global Transaction solution (as EJB are distributed components).

## 11.2 Local Transaction

Local Transaction is transaction which involves only one resource as part of transactional boundary. If you are working with database and message queue each and every operation that has been done are committed separately, rather than part of single transaction. The above figure depicts the same.

In a local transaction, the commit of a transaction will be done on the resource directly for example while you are working on JDBC, you will issue a commit on directly the connection using con.commit().

## 11.3 Benefit of Spring Transaction

By the above we understood that working with global and local transactions needs two different API's and need to learn two different API's. Your code is tightly coupled with one transaction implementation, and if you want to switch between local and global or with in global you need to modify your code. Instead spring transaction management capability has provided a unified approach that allows you to work with local or global transactions transparently.

Spring has provided three ways of working with transactions.

1) Programmatic transaction management
2) Declarative transaction management
3) Annotation driven transaction management

Out of which developers extensively uses declarative transaction management.

## 11.4 Declarative Transaction Management

In a declarative transaction management approach, the developer will declare the transactional semantics in a configuration file rather than programming it using code. The main advantage with this is instead of writing the logic to manage, your application classes are free from transaction related logic and if you want to switch or don't want to use transactions you don't need to modify your logic.

As transactions is a cross-cutting functionality, to implement transactionality we use declarative AOP configuration. In order to work with AOP based transactions, we need to import AOP and Tx Namespaces.

We begin and commit or rollback transactions on an operation, we will begin the transaction while entering a method and while returning from the method we will commit and if the method throws exception we will roll back the transaction. So, if you observe it is the combination of Around and Throws advice.

In order to manage it we need to declare an transactional advice which will manages the above said logic. In order to configure an transactional advice we need to use Tx namespace to declare it as shown below.

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
	<tx:attributes>
		<tx:method name="insert*" read-only="false"/>
	</tx:attributes>
</tx:advice>
```

The transaction advice will issue a commit or rollback on the transaction manager, so it needs a transaction manager to perform this, to declare a transaction manager; you need to pass the datasource as a reference to manage all the connections created by the datasource. read-only = "false" indicates you are performing an updatable operation. If you are performing only select type of operation you need to use read-only="true". Along with that you can specify the propagation attribute as well.

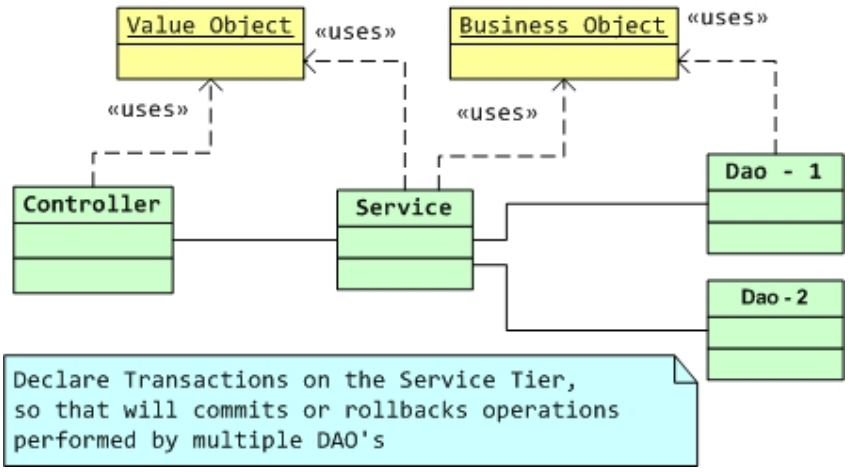
```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
	<property name="dataSource" ref="dataSource"/>
</bean>
```

Once the advice has been declared, you need to declare the classes on which you want to apply this advice using the <aop:confg> tag as shown below.

```
<aop:config>
	<aop:pointcut expression="execution(* com.dtx.service.*.*(..))" id="txpc"/>
	<aop:advisor advice-ref="txAdvice" pointcut-ref="txpc"/>
</aop:config>
```

## 11.5 Typical Spring Project Design

In a typical spring application, the transactionality will be imposed on Service tier classes, the same has been depicted in the below architecture.




Let us work on an example which follows the above architecture. In the below example we are trying to insert a Student, inserting a student involves inserting the Student information into student table as well as inserting into student course table. These two insertions has to be performed as part of single transactions. So the service will perform operations by using two dao's and the transaction is imposed on Service class.

**EducationalController.java**

```
package com.dtx.controller;

import com.dtx.service.DurgaEducationalService;

public class EducationalController {
	private DurgaEducationalService durgaEducationalService;

	public void insert(int studentId, String name, int courseId) {
		int outcome = 0;

		outcome = durgaEducationalService.insert(studentId, name, courseId);
		if (outcome > 0) {
			System.out.println("Student Inserted Successfully");
		}
	}

	public void setDurgaEducationalService(
			DurgaEducationalService durgaEducationalService) {
		this.durgaEducationalService = durgaEducationalService;
	}
}
```

**DurgaEducationalService.java**

```
package com.dtx.service;

public interface DurgaEducationalService {
	public int insert(int studentId, String name, int courseId);
}
```

**DurgaEducationalServiceImpl.java**

```
package com.dtx.service;

import org.springframework.transaction.annotation.Transactional;

import com.dtx.dao.StudentCourseDao;
import com.dtx.dao.StudentDao;

public class DurgaEducationalServiceImpl implements DurgaEducationalService {
	private StudentDao studentDao;
	private StudentCourseDao studentCourseDao;

	@Override
	public int insert(int studentId, String name, int courseId) {
		int outcome = 0;
		outcome = studentDao.insert(studentId, name);
		if (outcome > 0) {
			// re-intialize
			outcome = 0;
			outcome = studentCourseDao.insert(studentId, courseId);
		}
		return outcome;
	}

	public void setStudentDao(StudentDao studentDao) {
		this.studentDao = studentDao;
	}

	public void setStudentCourseDao(StudentCourseDao studentCourseDao) {
		this.studentCourseDao = studentCourseDao;
	}
}
```

**StudentDao.java**

```
package com.dtx.dao;

public interface StudentDao {
	public int insert(int studentId, String name);
}
```

**StudentDaoImpl.java**

```
package com.dtx.dao;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDaoImpl implements StudentDao {
	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentDaoImpl(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	@Override
	public int insert(int studentId, String name) {
		return jdbcTemplate.update(SQL_INSERT_STUDENT, new Object[] {
				studentId, name });
	}
}
```

**StudentCourseDao.java**

```
package com.dtx.dao;
public interface StudentCourseDao {
	public int insert(int studentId, int courseId);
}
```

**StudentCourseDaoImpl.java**

```
package com.dtx.dao;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentCourseDaoImpl implements StudentCourseDao {
	private final String SQL_INSERT_STUDENTCOURSE = "INSERT INTO
STUDENTCOURSE(STUDENT_ID, COURSE_ID) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentCourseDaoImpl(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	@Override
	public int insert(int studentId, int courseId) {
		return jdbcTemplate.update(SQL_INSERT_STUDENTCOURSE, new
Object[]{studentId,courseId});
	}
}
```

**application-context.xml**

```
<bean id="educationalController" class="com.dtx.controller.EducationalController">
	<property name="durgaEducationalService" ref="durgaEducationalService" />
</bean>

<bean id="durgaEducationalService"
class="com.dtx.service.DurgaEducationalServiceImpl">
	<property name="studentDao" ref="studentDao" />
	<property name="studentCourseDao" ref="studentCourseDao" />
</bean>

<bean id="dataSource"
	class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
	<property name="url" value="jdbc:oracle:thin:@//localhost:1521/xe" />
	<property name="username" value="hr" />
	<property name="password" value="hr" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
	<constructor-arg ref="dataSource" />
</bean>

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
	<property name="dataSource" ref="dataSource"/>
</bean>

<bean id="studentDao" class="com.dtx.dao.StudentDaoImpl">
	<constructor-arg ref="jdbcTemplate" />
</bean>

<bean id="studentCourseDao" class="com.dtx.dao.StudentCourseDaoImpl">
	<constructor-arg ref="jdbcTemplate" />
</bean>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
	<tx:attributes>
		<tx:method name="insert*" read-only="false"/>
	</tx:attributes>
</tx:advice>

 <aop:config>
	<aop:pointcut expression="execution(* com.dtx.service.*.*(..))" id="txpc"/>
	<aop:advisor advice-ref="txAdvice" pointcut-ref="txpc"/>
</aop:config>
```

## 11.6 Annotation approach Transaction Management

In this instead of declaring the transaction semantics in the declaration file we will directly annotate the class methods with @Transactional annotation. You can specify the readOnly and Propagation attributes in it.

In the earlier example instead of declaring the transactional semantics in the declaration, you can directly mark the service class methods with related to transaction annotation as shown below.

**DurgaEducationalServiceImpl.java**

```
package com.dtx.service;

import org.springframework.transaction.annotation.Transactional;

import com.dtx.dao.StudentCourseDao;
import com.dtx.dao.StudentDao;

public class DurgaEducationalServiceImpl implements DurgaEducationalService {
	private StudentDao studentDao;
	private StudentCourseDao studentCourseDao;

	@Override
	@Transactional(readOnly=”false”)
	public int insert(int studentId, String name, int courseId) {
		int outcome = 0;
		outcome = studentDao.insert(studentId, name);
		if (outcome > 0) {
			// re-intialize
			outcome = 0;
			outcome = studentCourseDao.insert(studentId, courseId);
		}
		return outcome;
	}

	public void setStudentDao(StudentDao studentDao) {
		this.studentDao = studentDao;
	}

	public void setStudentCourseDao(StudentCourseDao studentCourseDao) {
		this.studentCourseDao = studentCourseDao;
	}
}
```

In order to detect your annotations that has been marked at class level, you need to declare a tag in spring beans configuration file as

```
<tx:annotation-driven transaction-manager=”transactionManager”/>
```

# Spring MVC

# 12 Spring Web MVC (Model View and Controller)

Spring Web MVC Framework allows you to build web applications, similar to struts and J2ee servlets etc. When compared with any other frameworks, spring offers comprehensive list of features that makes you develop web applications with greater speed and flexibility.

## 12.1 Advantages of Spring Web MVC

1) Form Object: - In spring mvc you can use any object as command or form object (similar to Struts Action Form) to capture user submitted form values. Your Form object need not implement any spring specific classes or interface.

2) Flexible data binding: - Unlike other frameworks, spring form object can contain non-string typed as attributes, and spring will take care of binding the form submitted values to your form object attributes. In frameworks like struts, ActionForm should hold only string-typed attributes and need to perform the validation to capture type-casting errors. But spring will take care of these conversions and any in-compatible conversions will results into validation errors rather than runtime or type-casting errors.

3) Reusable business code: – no need of duplication, you can use your existing business objects as command or form objects.

4) Clear separation of roles: - controller, validator, command, handler mappings, dispatcher servlet, view resolvers, in this way spring provides multiple components to handle each role by a specialized object.

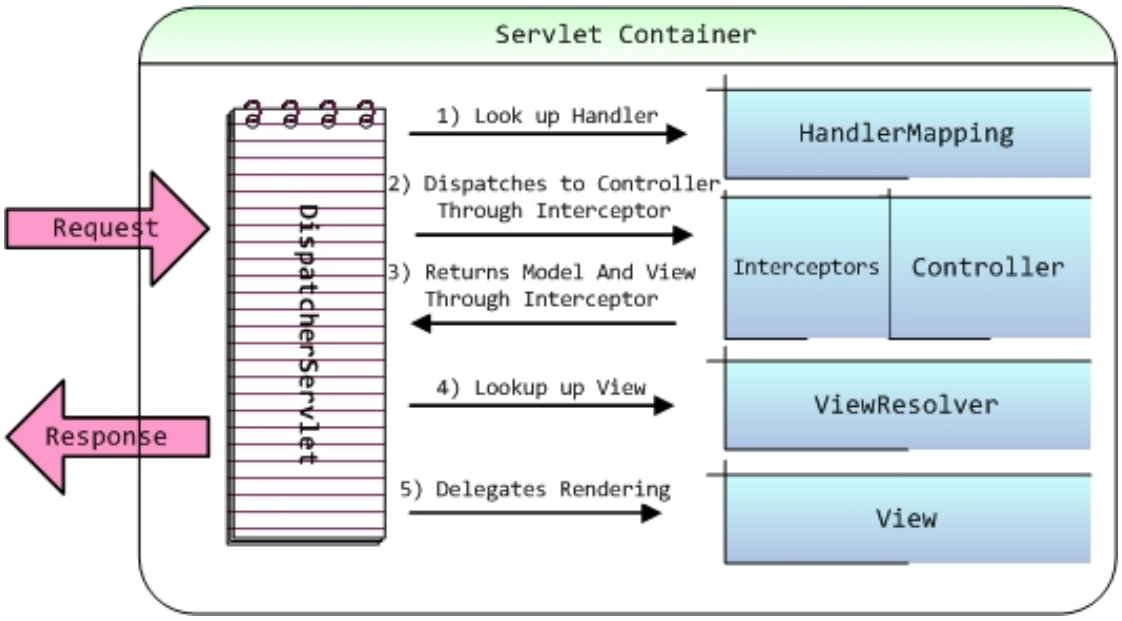
5) Locale and Theme resolvers: - spring provides jsp tag library that provides support for features such as data binding and themes.

## 12.2 Dispatcher Servlet

Spring Web MVC framework is designed around a DispatcherServlet that dispatches request to handlers, configurable handler mappings, view resolvers and locale and theme resolvers. Handler is the controller class in spring that will process the request and performs business logic to display the next view to the user. Handler mappings will allows you to map the incoming request to a handler. View resolvers will resolve for a view name, the physical view that should be rendered.

Spring Web MVC framework is, like any other web mvc frameworks, request-driven designed around a central servlet which is the DispatcherServlet. DispatcherServlet will listens for the incoming request and dispatches to handler, along with offers other functionalities facilitating the development of web application. In addition it completely integrates with IOC container making it to use every other feature of spring framework.

The typical request handling flow of the spring DispatcherServlet is shown below.




**Dispatcher Servlet Request Handling Flow**

Few points about Dispatcher Servlet as follows

1) DispatcherServlet acts as a FrontController – entry point for all the spring MVC requests
2) Loads XmlWebApplicationContext
3) Controls workflow and mediates between various MVC components
4) Loads sensible default components if none are configured

As every other servlet, even the DispatcherServlet has to be configured in the web.xml mapped to an URL to handle the incoming request, the mapping declaration is shown below.

**web.xml**

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

In the above configuration, all request's ending .htm will be handled by the DispatcherServlet. Each DispatcherServlet in spring has its own WebApplicationContext, a typical web application contains two types of components 1) Web components (controllers, command objects, view resolvers and views etc.) and 2) Business components (which performs business logic like Delegate, Service, Dao etc.), So the context that is getting created by DispatcherServlet will contains Web Components and Business components are declared separately which is going to be discussed shortly.

Spring while initializing the DispatcherServlet, will looks for the configuration file [servletname]-servlet.xml for the Web component bean declarations. Based on the above configuration, the file name it looks for is dispatcher-servlet.xml. Now loads all the bean declarations and creates a WebApplicationContainer with those beans.

Note: - You can customize the [servlet-name]-servlet.xml pattern by configuring a init-param at the DispatcherServlet where the param-name is namespace, value is the name of the configuration file shown below.

```
<servlet>
	<servlet-name>dispatcher</servlet-name>
	<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
	<init-param>
		<param-name>namespace</param-name>
		<param-value>ui</param-value>
	</init-param>
	<load-on-startup>2</load-on-startup>
</servlet>
```

*DispatcherServlet configuration with namespace*

With the above configuration, DispatcherServlet will look for the file ui.xml under WEB-INF directory.

## 12.3 Configuring ApplicationContext

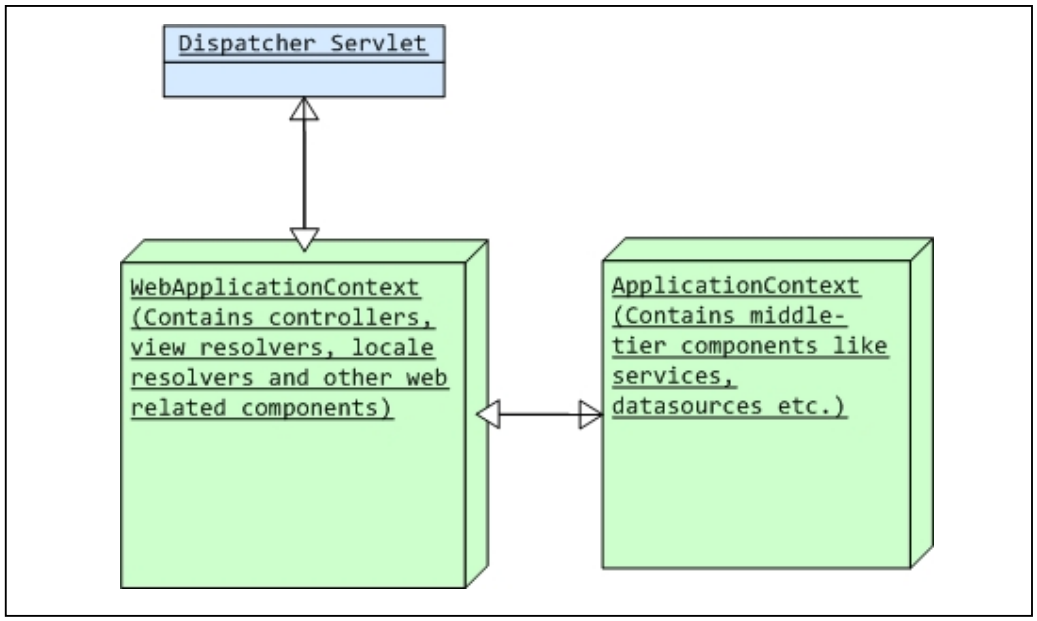
Apart from the WebApplicationContext created by the DispatcherServlet, you can configure one more container ApplicationContext which holds Business component bean declarations. In order to create this, you need to configure a Listener as shown below.

**web.xml**

```
<context-param>
	<param-name>contextConfigLocation</param-name>
	<param-value>/WEB-INF/root-application-context.xml</param-value>
</context-param>
<listener>
	<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

With the above configuration, the ContentLoaderListener will reads the context-param whose param-name is contextConfigLocation, and reads the value representing the configuration file path and creates the IOC container which contains the business component classes as beans.

Note: - The ApplicationContext created by ContextLoaderListener will act as parent factory to the WebApplicationContext which means the Web components can reference your business components whereas business components cannot refer your web components. If you recollect this is the concept of Nested Bean Factories, which we discussed in Spring Core. The main advantage of having two Containers is the web components and declarations are separated from business components, so that if you want to quit from Spring Web MVC, you need to remove the DispatcherServlet configuration and your Business components still can be injected into other mvc framework components through spring integration project.




*Diagrammatic representation of WebApplicationContext referencing ApplicationContext*

In order to handle the Web application requests, we need to use special components that are provided by mvc framework which are instantiated and mediated by DispatcherServlet (in WebApplicationComponents). In the following sections let's understand the components that must be written or configured to handle the MVC flow.

## 12.4 Controller

Controller is part of MVC design pattern (specifically it is 'C' in MVC). Controller handles the incoming request maps the request data to Object and sends that data to the Business tier (typically a Service class) classes to process the request. Spring has provided a wide variety of controllers to address different types of usecases like Simple Controllers, Form based controllers, Command-based controllers and Wizard-style logic controllers.

Spring basic controller architecture is org.springframework.web.servlet.mvc.Controller interface. The Controller interface has a single method as shown below.

```
ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception;
```

This method is responsible for handling the request and returns an appropriate model and view. The ModelAndView and Controller are basic concepts of spring MVC, based on this rest of the controllers has been designed.

Following is the list of controllers and the usecase in which we need to use.

| Controller Type | Description |
|---|---|
| AbstractController | Basic controller which implements from Controller Interface which contains infrastructure code for request handling |
| AbstractCommandController | This controller maps the incoming request values to the specified object (command object). |
| SimpleFormController | This controller not only maps the request data to an object along with that it supports form submissions and request validation |
| AbstractWizardController | This supports wizard based workflow management, where the request values will be carry forwarded and will be submitted at the end. |

### 12.4.1 Abstract Controller

When using AbstractController, you need to override only one method handleRequestInternal(HttpServletRequest, HttpServletResponse) method, implement your logic to return ModelAndView Object. Below code snippet shows how to work with AbstractController.

**StudentController.java**

```
package com.acweb.controller;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import com.acweb.business.StudentBO;
import com.acweb.service.StudentService;

public class StudentController extends AbstractController {
	private StudentService studentService;

	@Override
	protected ModelAndView handleRequestInternal(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		String name = request.getParameter("name");
		List<StudentBO> students = studentService.findStudent(name);

		return new ModelAndView("studentdetail", "students", students);
	}

	public void setStudentService(StudentService studentService) {
		this.studentService = studentService;
	}
}
```

**dispatcher-servlet.xml**

```
<bean id="studentDetailController" class="com.acweb.controller.StudentController">
	<property name="studentService" ref="studentService" />
</bean>
```

In the above example we have created the StudentController into which we injected a StudentService, where the StudentController is configured in dispatcher-servlet.xml as it is a Web Component and StudentService will be declared in application-context.xml which will be read by the ContextLoaderListener to create ApplicationContext.

### 12.4.2 Other simple controllers

Although you can extend the classes from AbstractController, spring has provided various concrete implementations which offer common functionality that is used commonly in simple MVC applications.

For example unlike in other MVC applications, spring MVC recommends to place all the jsp's or views that must be presented to the user under WEB-INF or sub-directories under it. This prohibits the users to directly access the views as WEB-INF and its sub-directories are private directories which are not directly exposed to the client. In order to access those jsp pages, the components of the application can only render those pages for example the controller can pull the jsp and can render it.

So in a simple usecase where you want to display a JSP page to the user, you need to write controller which returns the view name to be displayed, rather you writing the controller to perform this, spring has provided one of the implementation of Controller, which is ParameterizableViewController, this controller accepts a viewName as property and returns that configured logical view name to the ViewResolver. The following snippet shows the same.

```
<bean id="showStudentController"
class="org.springframework.web.servlet.mvc.ParametrizableViewController">
        <property name="viewName" value="student"/>
</bean>
```

In the above snippet the ParameterizableViewController returns the logical view name as student which you configured as property viewName in the configuration file.

There is another controller implementation UrlFilenameViewController, this inspects the URL and retrieves filename of the file from the request and uses that as the logical viewname.

```
<bean id="urlFileController"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

### 12.4.3 AbstractCommandController

This controller maps the user input request values to the specified command object. This controller has to be used only when we have simple forms where we don't need form handling capabilities, but want to map the request to command, then we need to use this.

You need to write a class which extends from AbstractCommandController and need to override the method handle, the command object is any pojo which holds the form values.

```
ModelAndView handle(HttpServletRequest, HttpServletResponse, Object command,
BindException) throws Exception
```

Along with this, you need to configure the pojo as command object at the controller configuration. Refer to the example below.

**StudentInsertController.java**

```
package com.acweb.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractCommandController;

import com.acweb.command.StudentCommand;
import com.acweb.service.StudentService;

public class StudentInsertController extends AbstractCommandController {
	private StudentService studentService;

	@Override
	protected ModelAndView handle(HttpServletRequest request,
			HttpServletResponse response, Object command,
			BindException exception) throws Exception {
		String message = null;
		StudentCommand scommand = (StudentCommand) command;
		int affectedRows = studentService.insertStudent(scommand);

		if (affectedRows > 0) {
			message = "Inserted Successfully";
		}
		return new ModelAndView("studentInsertConfirm", "msg", message);
	}

	public void setStudentService(StudentService studentService) {
		this.studentService = studentService;
	}

}
```

In the above class we are using the StudentCommand as command object.

**StudentCommand.java**

```
package com.acweb.command;

public class StudentCommand {
        private int studentId;
        private String name;

        // setters and getters
}
```

**dispatcher-servlet.xml**

```
<bean id="studentInsertController"
class="com.acweb.controller.StudentInsertController">
        <property name="studentService" ref="studentService" />
        <property name="commandClass"
value="com.acweb.command.StudentCommand" />
</bean>
```

In the above controller configuration, the Controller has been injected the property commandClass and the value as command class name, so that spring AbstractCommandController, will instantiates that object when you submit the form and populates the user entered field values into the attributes of the command object.

Note: - The HTML Control names and the command object attribute names must match to map their values automatically upon submitting the form.

**insertStudent.jsp**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
		<title>Insert Student</title>
	</head>
	<body>
		<form action="processInsertStudent.htm" method="POST">
			<table>
				<tr>
					<td>Id : </td>
					<td><input type="text" name="studentId"/></td>
				</tr>
				<tr>
					<td>Name : </td>
					<td><input type="text" name="name"/></td>
				</tr>
				<tr>
					<td colspan="2">
						<input type="submit" value="Insert"/>
					</td>
				</tr>
			</table>
		</form>
	</body>
</html>
```

### 12.4.4 SimpleFormController

SimpleFormController even provides more support for handling Form, it populates the form values into Command Object provided, along with that takes the view name for the form, and upon submitting the form will shows you the page upon success.

By the above, it is clear that it manages the entire form submission cycle, like displaying the initial page to render, once submitted the values will be populated to the command object and upon successful processing will renders a page.

In order to use a SimpleFormController, you need to extend a class from SimpleFormController and need to override two methods, formBackingObject(HttpServletRequest) and onSubmit(HttpServletRequest, HttpServletResponse, Object command, BindException);

The formBackingObject will be called while rendering the initial view, this method should return the command object populated with data, which you want to display while rendering the page. onSubmit method will be invoked when the form has been submitted. The onSubmit method has total three signatures as shown below.

```
onSubmit(Object command) throws Exception;
```

```
onSubmit(Object command, BindException) throws Exception;
```

```
onSubmit(HttpServletRequest, HttpServletResponse, Object command, BindException)
```

Out of the above three methods, you can decide to override any of the methods, incase if you have provided all the three methods, the max argument will be invoked on submitting the form.

**StudentController.java**

```
package com.sfw.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;

import com.sfw.command.Student;

public class StudentController extends SimpleFormController {

	@Override
	protected ModelAndView onSubmit(HttpServletRequest request,
			HttpServletResponse response, Object command, BindException
errors)
			throws Exception {
		Student sCommand = (Student) command;

		System.out.println("Id : " + sCommand.getId());
		return new ModelAndView("success");
	}

	@Override
	protected Object formBackingObject(HttpServletRequest request)
			throws Exception {
		return new Student();
	}

}
```

**dispatcher-servlet.xml**

```
<bean id="studentController" class="com.sfw.controller.StudentController">
        <property name="formView" value="insertStudent"/>
        <property name="commandClass" value="com.sfw.command.Student"/>
        <property name="validator" ref="studentValidator"/>
</bean>
```

In the above configuration, we have configured the property formView which has to be rendered when the SimpleFormController has first requested, while rendering the insertStudent view, it will call the formBackingObject to populate the data into the UI fields. Upon submitting the form, the onSubmit method will be called to process in the input values.

In the above xml, we have configured the validator which will be called to validate the form values, when you submitted the form and before the onSubmit method is invoked.

### 12.4.5 Validator

In order to create a validator you need to write a class which implements Validator interface and override two methods supports and validate method. Unlike struts, the validator component is the separate component, which will validate the command data, and you can attach multiple validators to a form. In order to ensure the validator is being invoked on the right command object, the supports method will perform this check, and returns Boolean. If the supports method returns true, then the validate method will be invoked on the command object otherwise will ignore.

**dispatcher-servlet.xml**

```
<bean id="studentValidator" class="com.sfw.validator.StudentValidator"/>

<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
</bean>
```

**messages.properties** *(Should be placed under src directory or its sub-directories)*

```
StudentId.Blank=Student Id is mandatory
StudentName.Blank=Student name is required
```

**StudentValidator.java**

```
package com.sfw.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.sfw.command.Student;

public class StudentValidator implements Validator {

	@Override
	public boolean supports(Class<?> classType) {
		return Student.class == classType;
	}

	@Override
	public void validate(Object command, Errors errors) {
		Student sCommand = (Student) command;

		if(sCommand.getId() <= 0) {
			errors.reject("StudentId.Blank");
		}

		if(sCommand.getName() == null || sCommand.getName().equals("")) {
			errors.reject("StudentName.Blank");
		}
	}
}
```

If the validator returns any errors, the relevant messages for the keys will be picked from the messageSource bean configured above and redisplays the form back to the user.

In order to re-display the form back to the user with user entered values, we need to write the JSP forms with spring form tag libraries. These libraries are similar to struts form tag libraries which allows you to map the user interface fields to the command attributes and help in displaying the form back up on errors with user entered values and error messages.

**insertStudent.jsp**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
		<title>Insert Student</title>
	</head>
	<body>
		<form:form method="POST">
			<table>
				<tr>
					<td colspan="2">
						<p style="color: red">
							<form:errors/>
						</p>
					</td>
				</tr>
				<tr>
					<td>Id</td>
					<td><form:input path="id"/></td>
				</tr>
				<tr>
					<td>Name</td>
					<td><form:input path="name"/></td>
				</tr>
				<tr>
					<td>
						<input type="submit" value="Insert"/>
					</td>
				</tr>
			</table>
		</form:form>
	</body>
</html>
```

## 12.5 Handler Mappings

Using handler mappings you can map an incoming web requests to appropriate handler. You can configure more than one handler mappings, based on the order (priority) will maps the request to a Handler (Controller). Basically it works in this way, the DispatcherServlet once receives the request will handover it to HandlerMapping to let it inspect the request and come up with HandlerExecutionChain. Then the DispatcherServlet will execute the handler in the chain.

HandlerMapping can optionally contain HandlerInterceptors, the concept of HandlerInterceptor will be discusses later, but the HandlerExcecutionChain returned will contain list of HandlerInterceptors and Handler for execution.

Two of the most commonly used Handler Mappings's in spring are discussed in the below section, but they both extend from a class AbstractHandlerMapping and share the properties as below.

1) Interceptors: - the list of interceptors to use. HandlerInterceptor will perform pre and post processing of request.

2) Order:- If multiple handler mappings has been configured then spring will sort all the handler mappings in the chain and will apply the first one in the chain.

### 12.5.1 BeanNameUrlHandlerMapping

This will maps the incoming request URL to the beanName to identify the controller for execution. Let's say for example we have an appropriate form controller which will handle the insert of an employee. When an incoming request likes /insertEmployee.htm come to the DispatcherServlet, it will forwards it to configured BeanNameUrlHanlderMapping to map it to the Controller, it will pick the controller bean whose name is "/insertEmployee.htm" as show below.

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean  name="/insertEmployee.htm"
class="com.emp.controller.InsertEmployeeController">
        <property name="formView" value="insertEmployee"/>
        <property name="successView" value="showEmployeeDetail"/>
        <property name="commandClass" value="com.emp.command.Employee"/>
</bean>
```

If you don't configure any handler mapping, by default spring will provide BeanNameUrlHandlerMapping as the default one.

### 12.5.1 SimpleUrlHandlerMapping

Much more powerful and most used handler mapping is SimpleUrlHandlerMapping. In this you will configure the mapping between the incoming requests to a handler in the configuration file. It allows you to map your request using ant-style path matching capabilities, the below configuration will lets you point to the InsertEmployeeController by using the url insert.htm.

```
<bean id=”handlerMapping”
class=”org.springframework.web.servlet.handler.SimpleUrlHandlerMapping”>
        <property name=”mappings”>
                <props>
                        <prop key=”/insert.htm”>insertEmployeeController</prop>
                </props>
        </property>
</bean>

<bean id=”insertEmployeeController”
class=”com.emp.controller.InsertEmployeeController”>
        <!—left for clarity →
</bean>
```

## 12.6 Handler Interceptors

In the HandlerMappings section we come across the HandlerIterceptors, which means the HandlerMappings along with mapping URL to a Handler will also configured with HandlerInterceptors. The main purpose of HandlerInterceptors is to handle pre and post processing of incoming request; this is similar to the concept of filters in J2EE. But the main difference between filters and interceptors is, filters are applied to all the requests of the web application, whereas interceptors are applied to certain group of handlers. Secondly you have three states of interceptor execution like before processing request, before rendering the view and after the view has rendered to the user.

In order to configure a class as HandlerInterceptor, you need to write a class which implements HandlerInterceptor. Instead you can override your class from HandlerInterceptorAdapter and decide to override the method of your choice as shown below.

**StoreTimeInterceptor.java**

```
package com.sw.handlerinterceptor;

import java.util.Calendar;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class StoreTimeInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request,
			HttpServletResponse response, Object handler) throws Exception {
		Calendar c = Calendar.getInstance();
		if (c.get(Calendar.HOUR) >= 5) {
			response.sendRedirect("timeout.jsp");
		}
		return true;
	}
}
```

**dispatcher-servlet.xml**

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
	<property name="mappings">
		<props>
			<prop key="/search.htm">SearchController</prop>
		</props>
	</property>
	<property name="interceptors">
		<list>
			<ref local="timeoutInterceptor"/>
		</list>
	</property>
</bean>

<bean id="timeoutInterceptor"
class="com.sw.handlerinterceptor.StoreTimeInterceptor"/>
```

## 12.7 ViewResolver

Every spring MVC framework will one way or in another provides a mechanism to dispatch the request to render a view. In Spring MVC it ViewResolvers are used for mapping and dispatching the request to an appropriate view.

When discussing about the controller, we observed that all the controllers will returns an ModelAndView as response after execution, the ModelAndView contains the logical view name that will be returned to the DispatcherServlet for rendering, the DispatcherServlet will maps the logical viewName to a View by using ViewResolver.

The two fundamentals of spring view handling is org.springframework.web.servlet.ViewResolver and org.springframework.web.servlet.View interfaces one will maps the view name to view and other prepares the request and handovers the request to render the View.

Spring has a rich set of pre-defined ViewResolvers provided out of box, which you can use straight forward.

| ViewResolver | Description |
|---|---|
| XmlViewResolver | An implementation of ViewResolver interface which accepts the XML file as View Configurations to resolve the views. |
| ResourceBundleViewResolver | Instead of using XML file for View configurations, you will use the properties file, the key is the logical view name and the value is the resource you want to render |
| UrlBasedViewResolver | A simple implementation of the ViewResolver interface which maps directly the symbolic view names to the URL's without explicit mapping definition |
| InternalResourceViewResolver | This is a sub class of UrlBasedViewResolver which supports direct rendering of InternalResourceViews like Servlets and JSP's. and other view Sub classes like JSTLView and TilesView |

An example of the ViewResolver configurations for each one is discussed in the following section.

### 12.7.1 UrlBasedViewResolver

This ViewResolver will maps the view name to a URL and handover's it to the DispatcherServlet to render the view.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
</bean>
```

### 12.7.1 ResourceBundleViewResolver

When working with different view technologies in a web application, you can use ResourceBundleViewResolver.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
        <property name="baseName" value="views"/>
</bean>
```

With the above declaration it means all the view definitions have been declared in the file views.properties under your classpath (src directory). The ResourceBundleViewResolver inspects the ResourceBundle and identifies the baseName for each view it is supposed to resolve.

The mapping information in the properties file would be like [viewname].class and [viewname].url, where [viewname].class represents the class that acts as view class to render the view and the [viewname].url represents the path to the view located in the application. Sample views.properties has been given below.

**views.properties**

```
insertStudent.class=org.springframework.web.servlet.view.JstlView
insertStudent.url=/WEB-INF/jsp/insertStudent.jsp
```

### 12.7.1 XmlViewResolver

Instead of providing the mappings in a properties file here we declare an xml file which contains bean definitions of the views that has to be render. The following configuration shows the same.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.XmlViewResolver">
        <property name="location" value="/WEB-INF/views.xml"/>
</bean>
```

In the above configuration, we indicated the view declarations are declared in views.xml

**views.xml**

```
<bean id="home" class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/home.jsp"/>
</bean>
```

# Spring ORM

# 13 Spring ORM (Object Relational Mapping)

Spring ORM framework allows you to integrate with Hibernate, Java Persistence API (JPA), Java Data Objects (JDO) and iBATIS for resource management and data access object (DAO) implementations and other transaction strategies.

Benefits of using Spring Framework to create your ORM DAOs include:

a) Easier testing: Spring IOC approach allows you to swap easily the implementations and configuration locations of Hibernate SessionFactory instances etc, so that you can point your configurations to various environments without modifiying the source code.

b) Common data acess exception: Spring instead of exposing ORM specific checked exceptions to the top level tier's of the application, it will wraps the technology specific exceptions to a common runtime DataAccessException hierarchy.

c) Integrated Transaction management: Instead of dealing with ORM technology related transactional code, it allows you to declaratively manage the transactionality using AOP Declarative transaction management tags <tx:advice> or annotation driven @Transactional annotation.

## 13.1 Integrating with Hibernate

In order to use spring ORM with hibernate, you need to declare your Hibernate Language objects (HLO) using declarative mapping.hbm files are annotate your classes with hibernate annotations. In order to perform the operations using these classes, you need to declare an HibernateSessionFactory and then injects it into HibernateTemplate, recall the concept of Template JDBCTemplate which allows you to perform the JDBC operations using Template approach.

Sample code is shown below.

**EmployeeHLO.java**

```
package com.ew.hlo;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "TBLEMP")
public class EmployeeHLO {
	private long id;
	private String name;
	private double salary;

	@Id
	@Column(name="EMP_ID")
	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
	}

	@Column(name="EMP_NM")
	public String getName() {
		return name;
	}

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

	@Column(name="SALARY")
	public double getSalary() {
		return salary;
	}

	public void setSalary(double salary) {
		this.salary = salary;
	}
}
```

**EmployeeDao.java**

```
package com.ew.dao;

import org.springframework.orm.hibernate3.HibernateTemplate;

import com.ew.hlo.EmployeeHLO;

public class EmployeeDao {
	private HibernateTemplate hibernateTemplate;

	public EmployeeDao(HibernateTemplate hibernateTemplate) {
		this.hibernateTemplate = hibernateTemplate;
	}

	public void insert(EmployeeHLO employeeHLO) {
		hibernateTemplate.save(employeeHLO);
	}
}
```

**EmployeeService.java**

```
package com.ew.service;

import com.ew.command.Employee;
import com.ew.dao.EmployeeDao;
import com.ew.hlo.EmployeeHLO;

public class EmployeeService {
	private EmployeeDao employeeDao;

	public void setEmployeeDao(EmployeeDao employeeDao) {
		this.employeeDao = employeeDao;
	}

	public void insert(Employee employee) {
		EmployeeHLO employeeHLO = new EmployeeHLO();
		employeeHLO.setId(employee.getId());
		employeeHLO.setName(employee.getName());
		employeeHLO.setSalary(employee.getSalary());
		employeeDao.insert(employeeHLO);
	}
}
```

**persistence-beans.xml**

```
<bean id="dataSource"
	class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<property name="driverClassName"
value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
	<property name="url"
		value="jdbc:sqlserver://localhost:1433;databaseName=spdb" />
	<property name="username" value="sa" />
	<property name="password" value="welcome1" />
</bean>

<bean id="sessionFactory"
	class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactor
yBean">
	<property name="dataSource" ref="dataSource" />
	<property name="annotatedClasses">
		<list>
			<value>com.ew.hlo.EmployeeHLO</value>
		</list>
	</property>
	<property name="hibernateProperties">
		<props>
			<prop
key="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</prop>
			<prop key="show_sql">true</prop>
		</props>
	</property>
</bean>

<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
	<property name="sessionFactory" ref="sessionFactory" />
</bean>
<bean id="transactionManager"
	class="org.springframework.orm.hibernate3.HibernateTransactionManager">
	<property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="employeeDao" class="com.ew.dao.EmployeeDao">
	<constructor-arg ref="hibernateTemplate"/>
</bean>
```

**aop-beans.xml**

```
<aop:config>
	<aop:pointcut expression="execution(* com.ew.service.EmployeeService.*(..))"
		id="empServicePC" />
	<aop:advisor advice-ref="empTxAdvice" pointcut-ref="empServicePC" />
</aop:config>
<tx:advice id="empTxAdvice" transaction-manager="transactionManager">
	<tx:attributes>
		<tx:method name="insert*" read-only="false" propagation="REQUIRED"
/>
	</tx:attributes>
</tx:advice>
```