Spring

Naresh i Technologies

Introduction:

- This course explains various utilities provided by Spring Framework. Spring is a lightweight open-source application framework that simplifies the enterprise application development in Java.
- In J2EE application development, we got some component technologies and service technologies.
- By using the component and service technologies of J2EE, we can develop a good enterprise level Business applications.
- Component Technologies are
  1. Servlet
  2. JSP
  3. EJB
- Service Technologies are
  1. JNDI (Java Naming and Directory Interface)
  2. JMS (Java Messaging Service)
  3. Java Mail
  4. Java Transaction Service (JTS)
  5. JAAS (Java Authentication And Authorizations Service)
- We use Servlet and JSP technologies of J2EE for the development of Web applications.
- We use EJB technology for the development of Distributed and Enterprise applications.

- While developing enterprise applications through EJB, we got the following problems.

  1. EJB Component development requires multiple Java files and xml files. So it makes burden on the developer or programmer.

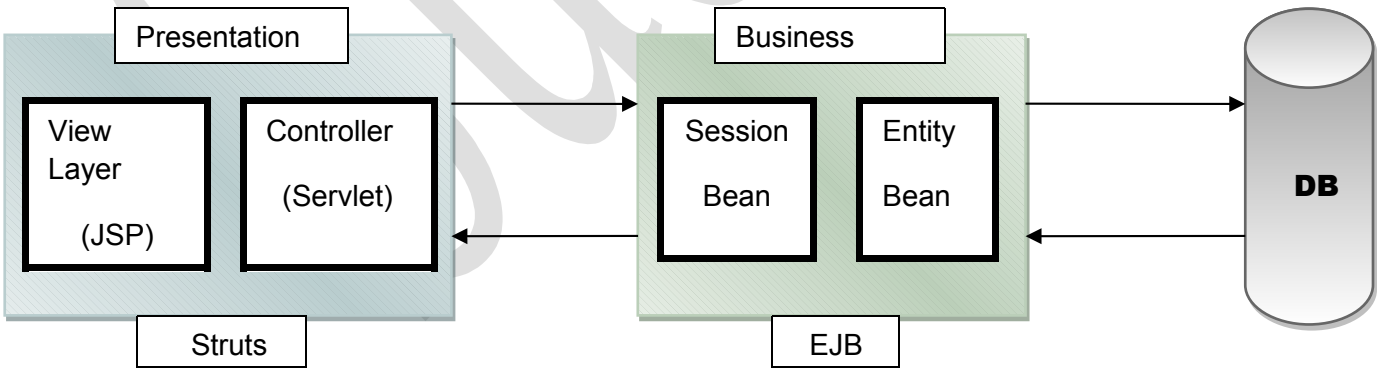
  2. EJB's cannot run without a container/Server. So to test an EJB a server is mandatory.

  3. While testing an EJB, if it files wants to be modified then we need to shutdown the server and after modification again we need to restart the server and test. If you want to test then we need to follow the same procedure.

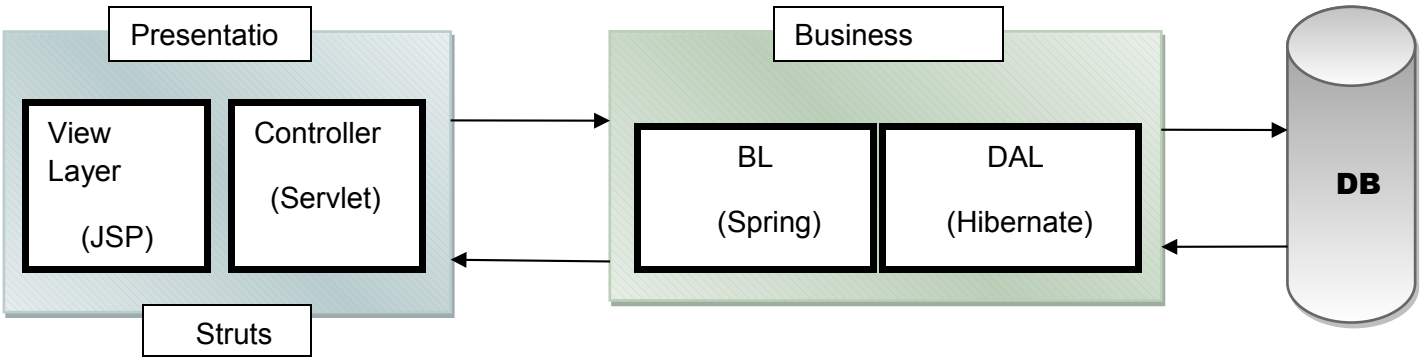
- EJB's are heavy weight components and having a lot of dependency with a server.

Architecture Diagrams:

1. ### Before Spring:-



2. ### After Spring:-



Spring Framework:

- Spring is a multi-tier open-source lightweight application framework, addressing most infrastructure concerns of enterprise application.
- It is mainly a technology dedicated to enable us to build applications using POJOs.
- Spring is a complete and a modular framework. It means Spring Framework can be used for all layers implementation for a Real-time applications (or) Spring can be used for the development of a particular layer of real time application.
- Spring Framework is said to be a non-invasive or non-intrusive framework. It means Spring Framework does not force a Programmer to extend or implement their classes from any pre-defined class or interface given by Spring API.
- In case of Struts Framework, it will force the programmer that the programmer class must extend from the base class provided by Struts API. So Struts is called as an invasive/intrusive Framework.

- Spring Framework simplified J2EE application development by introducing POJO model and is important because of following three reasons.

  1. Simplicity
  2. Testability
  3. Loose-Coupling

- Spring provides simplicity, because of POJI or POJO model and is easy to test, because Spring can be executed within a server and even without server.
- Spring provides loose-coupling, because of Inversion of Control mechanism.
- Spring Framework can be used to develop any kind of Java Application. It means we can develop starting from console application up to enterprise level application.
- Unlike other Frameworks, Spring Framework has its own container, So Spring runs even without Server also.

Benefits of Spring Framework

- Spring Framework addresses most of the infrastructure functionalities of the enterprise applications. This comes with number of benefits such as

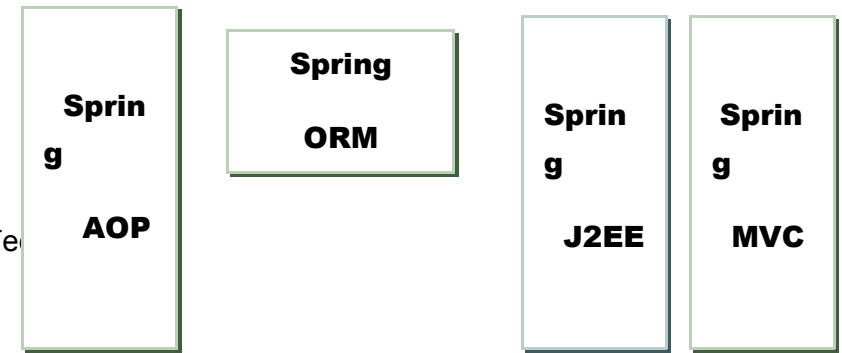
  1. Spring Framework address important areas that many other frameworks do not like. It includes support for managing business objects and exposing their services to the presentation tier components that the web and desktop applications can access the same business objects.

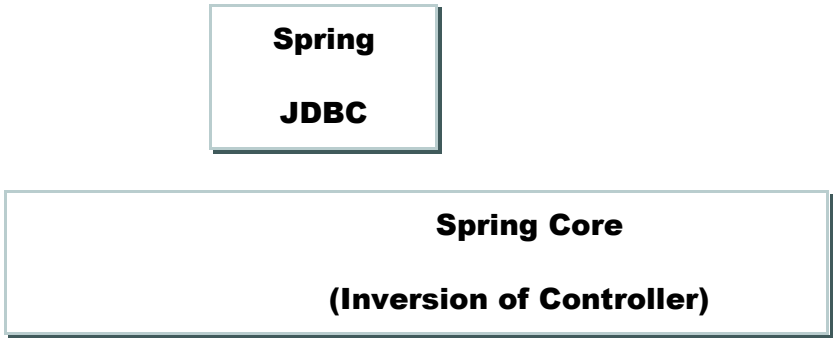
2. Spring is complete and modular. This is because Spring Framework has a layered architecture, meaning that you can choose to use just about any part of it in isolation, yet its architecture is internally consistent.
3. Spring Framework provides perfect support for write code that is easy to test. Meaning, Spring Framework is the best framework for test-driven projects.
4. Spring is an increasingly important integration technology; its role is recognized by several large vendors.
5. Spring can eliminate the creation of singletons and factory classes seen on many projects.
6. Spring can facilitates good programming practice by reducing the cost of programming interfaces, rather than classes, almost to zero

Spring Modules

- In Spring 1.x, the framework has divided into seven well-defined modules. But in Spring 2.x the framework is divided into 6 modules only.

1. Spring Core Module
2. Spring J2EE Module
3. Spring JDBC Module
4. Spring ORM Module
5. Spring AOP Module.
6. Spring MVC Module.





Spring Module Architecture

Spring Core Module:

- This module provides the basic functionality required for a Spring application
- Spring core module generates a high-level object of the Spring framework called “BeanFactory” or “ApplicationContext”.
- Spring core module applies IOC (Inversion of Control) or Dependency Injection for generating Spring IOC container called as BeanFactory.
- All frameworks will follows as an high level object is created first and it will have take care about the low level object required for the application.
- For Example

Struts Framework	-	ActionServlet
Hibernate Framework	-	SessionFactory
Spring Framework	-	BeanFactory

Spring Context (J2EE) Module:

- Spring J2EE module is an extension to core module where core module creates container and J2EE module make it as a framework.
- Spring J2EE module adds real-time services like Emailing,JMS,Timer,JNDI,Remoting and internationalization

Spring DAO module

- This module is providing an abstraction layer on top of existing JDBC technology and it avoids the repeated JDBC code from an application.
- Another advantage of this module is, we no need to handle exception while working with database. Because in Spring all exceptions are unchecked exceptions.

Spring ORM module

- This module is also work with database.
- Spring ORM module provides an abstraction layer on top of existing ORM tools.
- When compared with Spring DAO, Spring ORM has two benefits.
    a. Data transfer will be done in the form of Objects.
    b. An ORM tool always throws unchecked exceptions.
- Spring provides an abstraction layer on top existing ORM layer which avoids boils plate code.
- By without using this ORM modules, Spring can directly communicating with ORM tools like hibernate, iBATIS etc.

Spring AOP

- In real-time applications, business logic needs additional services.
- If we combinable implement business logic along with services, then we will get the following problems.
    a. The size of the application increases, so that complexity increases.
    b. If any modification is required in a service then in each business method we need to do modifications separately.
- We can avoid the above problems by using Aspect Oriented Programming in Spring.
- In AOP we can separated the business logic and secondary services.

Spring MVC Module

- In this module we can develop the following two operations.

a. We integrate our Spring framework with other frameworks like Struts and JSF etc.

b. We can develop complete MVC applications, by without using any other framework.

Spring Core Module (or) Spring IOC

Tight coupling and loose coupling between objects:

- While developing Java Applications one object will collaborate or communicate with another object for providing services to the clients.
- If one object is communicating with another object i.e. If one object is calling the business method of another object then we can say that there is a coupling between the two objects.
- In Object to Object collaboration, one object becomes dependent object and the other object becomes caller object.
- For example, If Traveler and Car are the two classes and Traveler class is depending on car. In this case Traveler Object is called dependent Object and that car Object is called Caller object.
- Coupling refers to dependencies that exist between two things. Consider two classes for an example. If one class is highly dependent on the implementation details of another class this is considered as tight coupling. Sometimes even both classes might be dependent on each other's implementation details making it more coupled. If a class is tightly coupled with another class, changing implementation details of main (depending) class will affect all other dependent classes.

```
public class Traveller {

	Car car = new Car();

	void startJourney()
	{
		car.move();
	}

}
```

```
public class Car {

	public void move() {

		//Business Logic
	}

}
```

- In the above example, Traveler Object is depending on Car Object. So Traveler class creating an object of Car class inside it.

- If directly the object is created in the dependent class then there exists tight coupling and the above example there is a tight coupling between Traveler and Car Object.
- In order to overcome tight coupling between objects, Spring Framework uses Dependency Injection Mechanism.
- With the help of POJO or POJI model and through Dependency Injection then it is possible to achieve loose coupling.
- In the above example, Traveler and Car are tightly coupled. If you want to achieve loose coupling between the objects Traveler and Car, we need to rewrite the application like the following.
- **Example:**

Vehicle.java

```
package com.nit.Spring.ioc;

public interface Vehicle {

	public void move();

}
```

Car.java

```
package com.nit.Spring.ioc;

/**
 * @author sudheer
 *
 */
public class Car implements Vehicle{

	@Override
	public void move() {

		// Business Logic

	}

}
```

Bike.java

```
package com.nit.Spring.ioc;

/**
 * @author sudheer
 *
 */
public class Bike implements Vehicle {

	@Override
	public void move() {

		// Business Logic
	}

}
```

Traveler.java

```
package com.nit.Spring.ioc;

/**
 * @author sudheer
 *
 */
public class Traveler {

	Vehicle vehicle;

	public void setVehicle(Vehicle vehicle) {
		this.vehicle = vehicle;
	}

	public void startJourney()
	{
		vehicle.move();
	}

}
```

- In the above example, through Dependency Injection mechanism, Spring container will inject either Car object or Bike Object into Traveler by calling setter method. So if Car object is replaced with Bike object then no changes are required in Traveler class. It means there is a loose coupling between Traveler Object and Vehicle Object.

Inversion of Control (IoC):

- Inversion of Control is an architectural pattern describing an external entity (that is container) used to wire objects at creation time by injecting there dependencies, that is, connecting the objects so that they can work together in a system.
- In other words the IoC describes that a dependency injection needs to be done by an external entity of creating the dependencies by the component itself.
- Dependency Injection is a process of injecting the dependencies in to an Object.

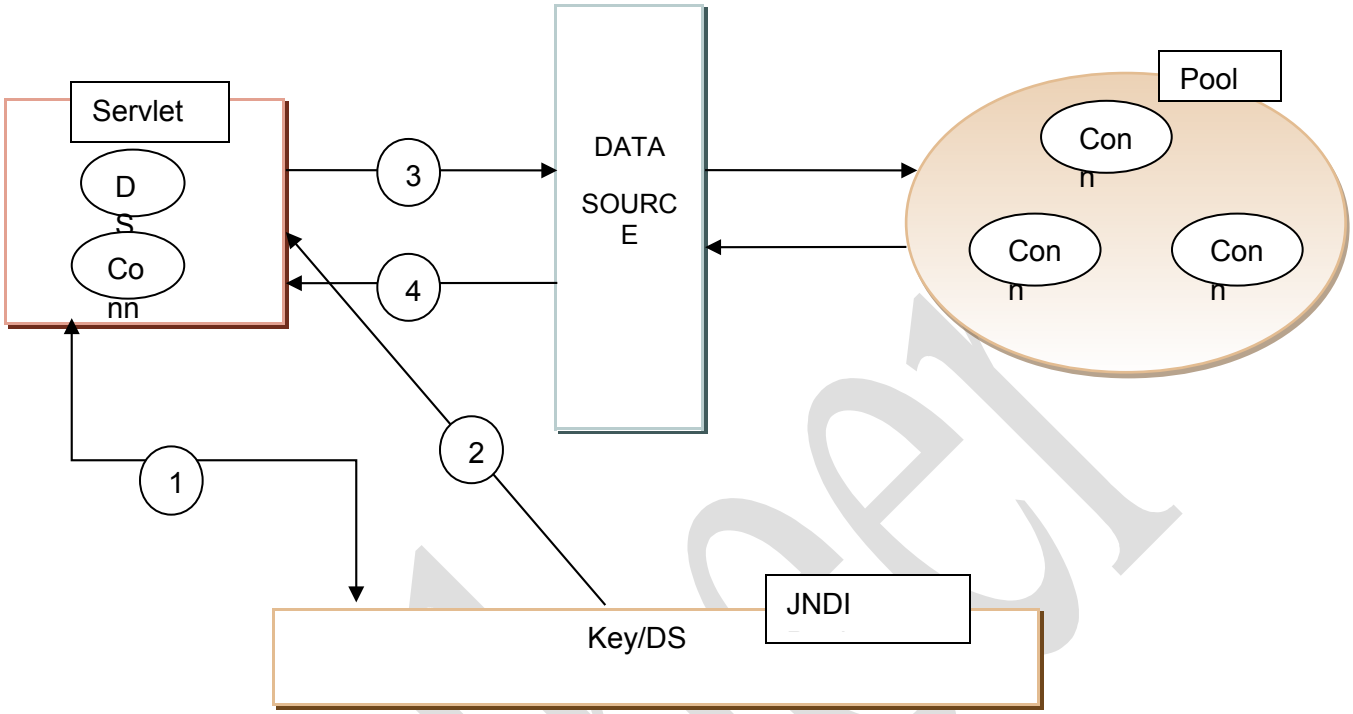
Types of IoC:

- Inversion of Control means, separating the implementation of an object from how the object is constructed.
- In case of Spring Framework the framework will take care of about the object creation and injecting the dependent object required. As a programmer we are responsible for only implementation of the business logic.
- There are two types of IoC's
  a. Dependency Lookup
  b. Dependency Injection

Dependency Lookup

- In this type of IoC, when one object is collaborating with another object, the first object has to get all the dependencies required explicitly by itself.
- In Dependency Lookup, an external person does not provide collaborating objects automatically to first object.
- In dependency lookup of IoC, the first object is responsible for obtaining its collaborating objects from the container or registery etc.
- For Example:
  a. In Servlet with Connection pooling, Servlet object depends on datasource object. In this case nobody automatically gives DataSource object to Servlet

object. Servlet itself goes to JNDI registry and takes DataSource object from the registery by performing lookup operation.



b. In Servlet to EJB communication, Servlet depends on EJB Object, For calling the business logic implemented in an EJB. In this communication nobody will provide EJB object to Servlet object automatically. Servlet itself goes to EJB container and gets the required EJB object from it. It means there is a Dependency Lookup.

Dependency Injection:

- In this type of IoC, when one object is collaborating with another object, some other external person will provide the collaborating object to the dependent object automatically.
- In this type of IoC the first object is not responsible to explicitly take the collaborating objects required. It means the first object does not perform any lookup operation.
- In case of Spring Framework, the external person who performs dependency injection is called Spring IoC container.

- EJB 3.0 Technology and Spring Framework both are supporting Dependency Injection type of IoC.

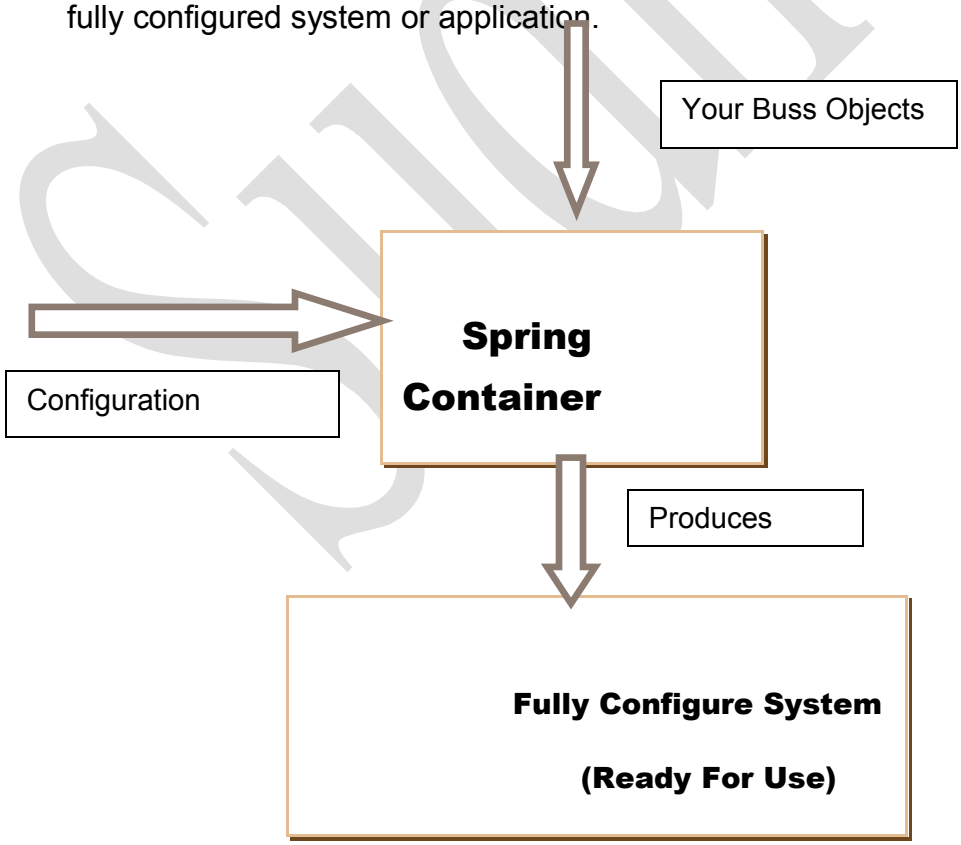
Why Dependency Injection

- In developing huge systems using the object oriented programming methodology, we generally divide the system into objects where each of the objects representing some functionality.
- In this case, the objects in the system use some other objects to complete the given request. The objects with which our object collaborates to provide the services are known as its dependencies.
- The traditional ways of obtaining the dependencies are by creating the dependencies or by pulling the dependencies using some object factory classes and methods or from the naming registery. But these approaches result in some problems which are described below:
  i. The complexity of the application increases.
  ii. The development time-dependency increases.
  iii. The difficulty for unit testing increases.
- To solve the above stated problems we have to use a push model, that is, inject the dependent objects into our object instead of creating or pulling the dependent objects. The process of injecting (pushing) the dependencies into an object is known as Dependency Injection (DI) .This gives some benefits as described below.
  i. The application development will become faster.
  ii. Dependency will be reduced.

Spring Core Container and Beans

- Spring core container is the basis for the complete Spring Framework. It provides an implementation for IoC supporting dependency injection.
- This provides a convenient environment to program the basic to enterprise application avoiding the need of writing the factory classes and methods in most of the situations.
- This even helps one to avoid the need of programming in singletons.

- The Spring core container takes the configurations to understand the bean objects that it has to instantiate and manage the lifecycle.
- The one important feature of the Spring core container is that it does not force the user to have any one format of configurations, as most of the framework which are designed to have XML based or Property file base configuration, or programmatically using the Spring beans API. But in most of the cases we use XML based configurations since they are simple and easy to modify.
- The org.springframework.beans.factory.BeanFactory is the actual representation of the Spring IoC container that is responsible for containing and otherwise managing the aforementioned beans.
- The BeanFactory interface is the central IoC container interface in Spring. Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.
- There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the XmlBeanFactory class. This implementation allows you to express the objects that compose your application, and the doubtless rich interdependencies between such objects, in terms of XML. The XmlBeanFactory takes this XML configuration metadata and uses it to create a fully configured system or application.



Configuration Metadata

- As can be seen in the above image, the Spring IoC container consumes some form of configuration metadata; this configuration metadata is nothing more than how you (as an application developer) inform the Spring container as to how to “instantiate, configure, and assemble [the objects in your application.
- This configuration metadata is typically supplied in a simple and intuitive XML format. When using XML-based configuration metadata, you write bean definitions for those beans that you want the Spring IoC container to manage, and then let the container do it's stuff.
- Spring configuration consists of at least one bean definition that the container must manage, but typically there will be more than one bean definition. When using XML-based configuration metadata, these beans are configured as <bean/> elements inside a top-level <beans/> element.
- Find below an example of the basic structure of XML-based configuration metadata.

```
<?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-2.5.xsd">
<bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
</bean>
        <bean id="..." class="...">
                <!-- collaborators and configuration for this bean go here -->
        </bean>
        <!-- more bean definitions go here... -->
</beans>
```

Instantiating a container:

- The Spring core container can be instantiated by creating an object of any one of the BeanFactory or ApplicationContext classes supplying the Spring beans configurations.

- The various implementation of BeanFactory are mainly differentiated based on the Spring beans configuration format they understand and the way they locate the configurations.
- The following code snippet shows the sample code that instantiates the Spring container using the Spring Beans XML configuration file as configuration metadata.
- **Syntax:**

  BeanFactory beans = new XmlBeanFactory (new FileSystemResource ("beans.xml"));
- In the preceding code snippet we are using XmlBeanFactory implementation for instantiating the Spring container with "beans.xml" file as a configuration file, as the name describes the ClassPathXmlApplicationContext locates the XML configuration file in the classpath
- We can even use ApplicationContext to instantiate the container; the following code snippet shows the sample code to instantiate Spring.
- **Syntax**

  ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
- In code snippet we are using ClassPathXmlApplicationContext implementation to instantiate the Spring container with "beans.xml" file as a configuration file, as the name describes the ClassPathXmlApplicationContext locates the XML configuration file in the classpath.

Types of Dependency Injection:

- Dependency Injection classified as the following three types.

  i. Setter Injection

  ii. Constructor Injection

  iii. Interface Injection

- In Spring Framework we use Setter and Constructor Injection but not Interface Injection.
- In Struts 2.X, we have Interface Injection.

Setter Injection

- In this type of Dependency Injection, the Spring (IoC) container uses setter method in the dependent class for injecting its dependencies (Collaborators).
- Spring Container knows whether to perform Setter Injection or Constructor Injection by reading information from an external file called as Spring Configuration file.
- In case of Setter Injection, the class must contain a setter method to get the dependencies; otherwise Spring container does not inject the dependencies to the dependent object.
- **Example**

```
/**
 * @author sudheer
 *
 */
public class A {

	B obj;

	public void setObj(B obj) {
		this.obj = obj;
	}

	public void m1()
	{
		// Business Logic

	}


}
```

```
public class B {

	// Business Logic

}
```

- In the above example, Class A is called Dependent and Class B is called Collaborator.
- In the Dependent class there is a setter method for getting collaborator object. So we call it as Setter Injection.
- In Spring Framework, we call each class as a “Spring Bean” .This Spring Bean is no where related with Java Bean.
- Spring Bean and Java Bean both are not same because a Java bean needs definitely a public default constructor. But in a Spring bean sometimes we include default constructor and sometimes we do not.
- In Spring bean classes; there exists the following three types of dependency values.
    i. Dependency in the form of Primitive Values
    ii. Dependency in the form of Objects
    iii. Dependency in the form of Collections

```
public class TestBean {

	// Dependency in the form of Primitive Values

	private int i ;

	// Dependency in the form of Objects

	private SampeleBean sampleBean;

	// Dependency in the form Collections

	private List list;

}
```

Dependency in the form of Primitives

- The Spring container understands whether dependency is in the form of primitive or not, whether setter injection or constructor injection is required information by reading Spring configuration file.
- Spring Configuration file is identified with <any name>.xml
- If the dependency in the form of primitives then we can directly provide value in the xml file for that property.
- In Spring configuration file, we used <property> element. So Spring container understands that there is a Setter Injection in the bean class.
- Inside <property> element, we have used <value> element. So Spring container understands that the dependency is in the form of Primitives.

  Syntax:

  <property name="k">

  <value>100</value>

  </property>

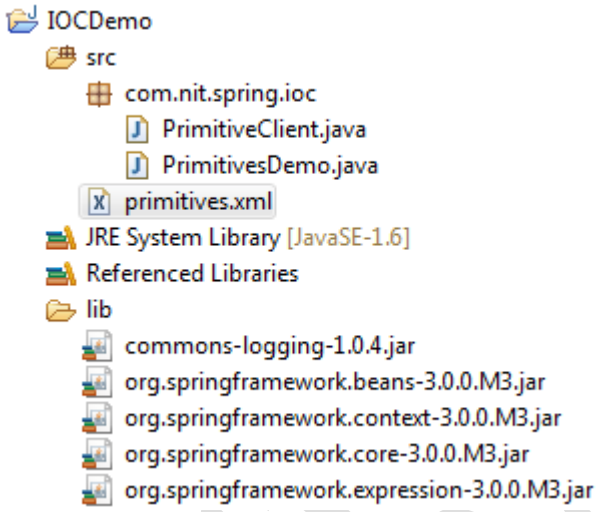
- In Spring Configuration file, we can use <value> as a sub element of <property> tag or value as an attribute of <property> tag

  Syntax:

<property name="k" value="100"/>

- **Example**

a. Project Structure



b. PrimitivesDemo.java

```java
package com.nit.spring.ioc;

/**
 * @author sudheer
 *
 */
public class PrimitivesDemo {

	String name;
	int age;

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

	public void setAge(int age) {
		this.age = age;
	}


	public void getDetails()
	{

		System.out.println(" Name:: "+name);

		System.out.println(" Age:: "+ age);
	}

}
```

C. PrimitivesClient.java

```java
package com.nit.spring.ioc;

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

public class PrimitiveClient {

	public static void main(String args[])
	{
		Resource res = new ClassPathResource("primitives.xml");

		BeanFactory beanF = new XmlBeanFactory(res);

		PrimitivesDemo pd = (PrimitivesDemo)beanF.getBean("pd");

		pd.getDetails();
	}

}
```

c. Primitives.xml (Configuration File)

```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="pd" class="com.nit.spring.ioc.PrimitivesDemo">
              <property name="name">
                     <value>Sudheer</value>
              </property>

              <property name="age">
                     <value>26</value>
              </property>

       </bean>

</beans>
```

d. **Output**

```
Name:: Sudheer
Age:: 26
```

Description of PrimitivesClient

Step 1:

A. Spring environment starts by loading Spring Configuration file into Resource Object.

B. We call this Stpe1 as a bootstrapping of Spring Framework

C. Resource is an interface and classpath. Resource is an implementation class given by Spring Framework.

D. Both Resource interface and classpath Resource class are given in ***org.springframewok.core.io*** package

Syntax:

Resource resource = new ClassPathResource ("beans.xml");

Step 2:

A. Spring IoC container object will be created by reading bean definitions from the Resource object.

B. Spring IoC container is called BeanFactory and this container is responsible for creating the bean objects and for assigning its dependencies.

C. BeanFactory is an interface and XmlBeanFactory is an implementation class of it.

D. BeanFactory is an interface given in ***org.springframework.beans.factory*** and XmlBeanFactory is a class given in ***org.springframework.beans.factory.xml*** package

Syntax:

BeanFactory beanFactory = new XmlBeanFactory(res);

Step 3:

A. Get the bean object from the Spring container by calling getBean() method.

B. While calling getBean() method, we need to pass the bean id as a parameter.

C. getBean() always return object.

D. We need to type cast the object into our Spring Bean type.

Syntax:

Object obj = beanFactory.getBean("pd");

PrimitivesDemo pd = (PrimitivesDemo)obj;

E) Call the business method of the SpringBean by using the object.

Pd.getDetails ()

Dependencies in the form of Objects

- While constructing Spring Beans (Pojo Classes), one Spring bean class depends on another Spring Bean class, for performing some business operations.

- If one bean class is depending on another bean class object then we call it as an object dependency.
- If one bean class is depending on another bean class object then the Spring IoC container is responsible for creating and injecting the dependencies.
- In Spring Configuration file we have two ways to inform the container about this object dependency.

  1. By using inner beans.
  2. By using <ref> element.

By using inner beans

- Inner bean means a bean which is added for a property by without an id in the xml file.
- In case of inner bean definition with setter injection, we should add the <bean> element, inside the <property> tag in xml.

```
/**
 *
 */
package com.nit.spring.ioc.object;

/**
 * @author Administrator
 *
 */
public class ObjectDemo1 {

	ObjectDemo2 od;

	public void setOd(ObjectDemo2 od) {
		this.od = od;
	}

	public void getDetails()
	{
		od.getObject();
	}

}
```

```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">

       <!-- Inner Bean Configuration -->

       <bean id="od1" class="com.nit.spring.ioc.object.ObjectDemo1">
              <property name="od">
                     <bean class="com.nit.spring.ioc.object.ObjectDemo2" />
              </property>
       </bean>

</beans>
```

- In the client application, if we call factory.getBean("od1") then internally the Spring Framework will do the following operations.

ObjectDemo1 od1 = new ObjectDemo1 ();

ObjectDemo2 od2 = new ObjectDemo2();

od1.setOd(od2);

- In the above example, ObjectDemo1 object is depending on ObjectDemo2 object. So the Spring container first created the ObjectDemo2 object and after that the container created ObjectDemo1 object.

Drawbacks of InnerBeans

- An Inner bean does not have any id. So it is not possible to get that bean object individually from the IoC container.

- If another bean class is also depending on the same bean then in the xml file , again we need to add the inner bean definition.
- In order to overcome the above two problems of inner beans, we need to use <ref> element in the Spring configuration file.

<ref> element

- When dependencies are in the form of objects then to inform the IoC container, in Spring configuration xml file, we need to configure <ref> element.
- <ref> element is associated with either local or parent and bean attributes.
- When we add <ref> element then we need to pass id of collaborator bean to its attribute, because the dependency is in the form of objects.

Local

- If local attribute is used with the <ref> element then the Spring IoC container will verify for the collaborator bean within same container.
- In general, we try to configure all Spring beans into a single Spring configuration file. But it is not mandatory, because we can create multiple Spring configuration files also.
-

```java
/**
 *
 */
package com.nit.spring.ioc.object;

/**
 * @author Administrator
 *
 */
public class ObjectDemo1 {

	ObjectDemo2 od;

	public void setOd(ObjectDemo2 od) {
		this.od = od;
	}

	public void getDetails()
	{
		od.getObject();
	}

}
```

```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="od1" class="com.nit.spring.ioc.object.ObjectDemo1">
		<property name="od">
			<ref bean="od2" />
		</property>
	</bean>

	<bean id="od2" class="com.nit.spring.ioc.object.ObjectDemo2" />

</beans>
```

- In the above xml file, both the dependent bean and its collaborator bean are configured in the same xml file. It means into the same Spring IoC container. So we can use local attribute with <ref> element.
- By loading this above xml file, we will get the Spring IoC container object called BeanFactory.

Syntax

Resource resource = new ClassPathResource("spconfig.xml");

BeanFactory beanFactory = new XmlBeanFactory (resource, null);

- In Spring IoC, it is possible to create parent and child factories.

Syntax

Resource resource = new ClassPathResource("parent.xml");

BeanFactory beanFactory = new XmlBeanFactory (resource, null);

Resource resource1 = new ClassPathResource ("child.xml");

BeanFactory beanFactory 1= new XmlBeanFactory (resource1, beanFactory);

- In the above syntax, beanFactory1 is a child container of beanFactory.
- DependentBean is available in beanFactory (Parent Container). It means both dependent bean and its collaborator bean are not available in same container. So we cannot use local attribute with <ref> element.

Parent

- If parent attribute is used with <ref> element then the Spring IoC container will search for the collaborator bean always at parent container. But not in the Same container.

parent.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="od2" class="com.nit.spring.ioc.object.ObjectDemo2"/>

</beans>
```

child.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="od1" class="com.nit.spring.ioc.object.ObjectDemo1">
              <property name="od">
                     <ref parent="od2" />
              </property>
       </bean>

</beans>
```

- In the above xml files, dependent bean is available in beanFactory1 (child container) and collaborator bean is available in beanFactory (parent container). So parent attribute is suitable with <ref> element.

Bean

- In this attribute is used with <ref> element then the Spring IoC container first verifies for the collaborator bean in the same factory. If not available then it will search in the parent factory.
- Bean is the combination of both local and parent.
- Bean first works like local and otherwise it works like parent.

parent.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="od2" class="com.nit.spring.ioc.object.ObjectDemo2"/>

</beans>
```

Child.xml

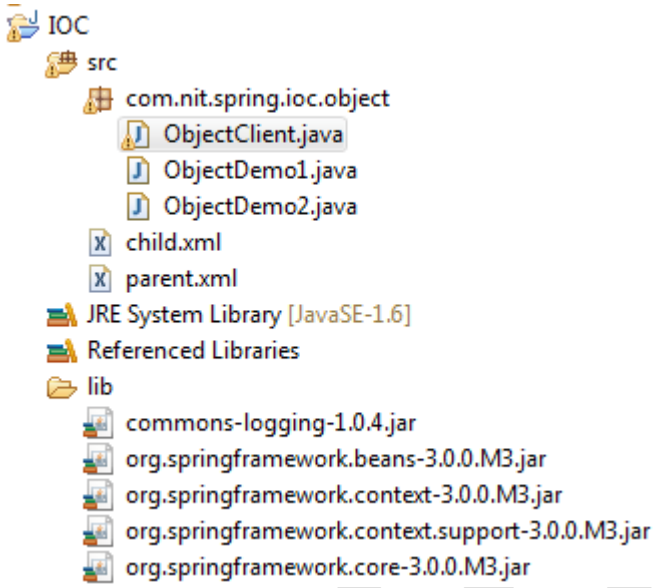
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">


       <bean id="od1" class="com.nit.spring.ioc.object.ObjectDemo1">
              <property name="od">
                     <ref bean="od2" />
              </property>
       </bean>

</beans>
```

Note:

1. While using <ref> element, if the given id is not found then null will be assigned into the object, but an exception is not thrown.
2. If id is not found but class is not a suitable for the required type then an exception will be thrown by the IoC container. The exception name is *org.springframework.beans.UnsatisfiedDependencyInjectionException.*

Example:

1. Project Structure



ObjectDemo1.java

```
package com.nit.spring.ioc.object;

/**
 * @author Administrator
 *
 */
public class ObjectDemo1 {

	ObjectDemo2 od;

	public void setOd(ObjectDemo2 od) {
		this.od = od;
	}

	public void getDetails()
	{
		od.getObject();
	}

}
```

ObjectDemo2.java

```
package com.nit.spring.ioc.object;

/**
 * @author Administrator
 *
 */
public class ObjectDemo2 {

	public void getObject()
	{
		System.out.println(" Dependencies in the
form Objects");
	}

}
```

ObjectClient.java

```java
package com.nit.spring.ioc.object;

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

/**
 * @author Administrator
 *
 */
public class ObjectClient {


	public static void main(String args[])
	{


	 Resource res1 = new ClassPathResource("parent.xml");

		BeanFactory bf1 = new XmlBeanFactory(res1,null);

		Resource res = new ClassPathResource("child.xml");

		BeanFactory bf = new XmlBeanFactory(res,bf1);

		ObjectDemo1 od1 = (ObjectDemo1)bf.getBean("od1");

		od1.getDetails();

	}

}
```

Parent.xml

```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="od2" class="com.nit.spring.ioc.object.ObjectDemo2"/>

</beans>
```

Child.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">

       <!-- Inner Bean Configuration -->
       <!--<bean id="od1"
class="com.nit.spring.ioc.object.ObjectDemo1">
           <property name="od">
             <bean class="com.nit.spring.ioc.object.ObjectDemo2"/>
            </property>
             </bean>
        -->

       <bean id="od1" class="com.nit.spring.ioc.object.ObjectDemo1">
             <property name="od">
                    <!--
                    <ref local="od2"/>
                    <ref parent="od2"/>
                  -->
                    <ref bean="od2" />
             </property>
       </bean>
       <!-- <bean id="od2"
class="com.nit.spring.ioc.object.ObjectDemo2"/> -->
</beans>
```

Output:

```
Dependencies in the form Objects
```

Dependencies in the form of Collections

- While creating a Spring bean (Pojo) , the bean class can use any one of the following four types of collections as a dependency.

  1. Set - <set>
  2. List - <list>
  3. Map - <map>
  4. Properties - <props>

- Except the above four types of collections, if the Spring bean class uses any other type of collection as a dependency then the Spring container does not inject that collection object to the Spring bean. In this case Spring programmer is responsible for injecting the collection object manually.

Set Collection

- If a Spring bean has a property of collection type set then in the Spring configuration file we need to use <set> element to inform the Sprig IoC.
- In Spring configuration file we can use <value> and <ref> tags as a sub elements of <set> tab.
- While configuring <set> element in the xml file it does not allow duplicate values. Because set collection is unique collection.
- In Spring Framework if one bean class is collaborating with another bean class then Spring IoC container first creates collaborator bean object and after that dependent bean object

Spring JDBC

1. Introduction:

- JDBC technology is required either directly or indirectly for getting a connection with the database.

- Without using JDBC technologies we cannot able connect with databases using Java.

- If a programmer is directly working with JDBC technology then there are some problems facing by the Java Programmer.

  a. JDBC technology exceptions are checked exceptions. There we must put try and catch blocks in the code at various places and it increases the complexity of the applicationcations.

  b. In a Java Program, repetitive code is required to perform operations on databases from the various client applications. The repetitive code is like loading the drivers, connection opening, creating a statement and finally closing the objects of JDBC. This kind of repetitive code, we call as Boilerplate Code.

  c. In JDBC, if we open the connection with database then we are responsible to close it. If the connection is not closed then later at some point of time, we may get out of connections problem.

  d. JDBC throws error codes of the database when an exception is occurred sometimes the error codes are unknown of the Java Programmer and error codes are different from one database to another database. Therefore developing JDBC applications using those error codes will make our application as database dependent.

- Spring-Jdbc frame work has provided an abstraction layer on top of the existing JDBC technology.

- Spring-Jdbc layer concentrates on avoiding connection management coding and error management and Spring-Jdbc programmer will concentrate on construction and execution of the SQL operations.

- Spring framework has provided an exception translator and it translates the checked exceptions obtained while using JDBC into unchecked exceptions of

Spring type and finally the unchecked exceptions are thrown to the Java Programmer.

- While working with Spring-Jdbc, the programmer no need to open and close the database connection and it will taken care by the Spring framework.

Table 1.1

The following table describes which actions is developer's responsibility and which actions are taking care by Spring itself:

Sno	Action	Developer	Spring
1	Define parameters for connection	Y	
2	connection opening		Y
3	Specify the statement for SQL	Y	
4	Declaration of parameters & providing value for parameter.	Y	
5	Prepare Statement & execute		Y
6	Loop setup for result iteration(if required)		Y
7	Defining work for each iteration	Y	
8	exception handling		Y
9	Handling transactions		Y
10	Connection, statement & resultset closing		Y

2. The package hierarchy

- The Spring Framework's JDBC abstraction framework consists of four different packages, namely core, datasource, object, and support.
- The ***org.springframework.jdbc.core*** package contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes.
- A sub-package named ***org.springframework.jdbc.core.simple*** contains the SimpleJdbcTemplate class and the related SimpleJdbcInsert and SimpleJdbcCall classes.

- Another sub-package named ***org.springframework.jdbc.core.namedparam*** contains the NamedParameterJdbcTemplate class and the related support classes.
  The ***org.springframework.jdbc.datasource*** package contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a J2EE container. The utility class provides static methods to obtain connections from JNDI and to close connections if necessary. It has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.
- Next, the ***org.springframework.jdbc.object*** package contains classes that represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects. This approach is modeled by JDO, although of course objects returned by queries are "disconnected" from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package.
- Finally the ***org.springframework.jdbc.support*** package is where you find the SQLException translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked giving you the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller.

3. DataSource:

- A Java application can get a connection with a database using the following two ways.

  i. *java.sql.DriverManager* (class)

  ii. *javax.sql.DataSource* (Interface)

- Spring framework always uses DataSource interface to obtain a connection with the database internally.

- DataSource interface is also having different types of implementation.

  i. Basic Implementation, it is equal to DriverManager.

  ii. Connection pooling implementation.

  iii. Transaction implementation.

- While using DataSource, it is always does not mean that we are working with connection pool. If internal implementation class is connection pool enabled then only the DataSource will provide a logical connection from the pool.
- In Spring framework we use any one of the following two implementation classes of DataSource interface
  i. *org.springframework.jdbc.datasource.DriverManagerDataSource*
  ii. *org.springframework.jdbc.datasource.BasicDataSource*
- The above two classes are suitable, when our Spring application is under development mode.
- In Production mode, the Spring application uses connection pooling service provided by the application servers.
- In the above two classes, DriverManagerDataSource is given by Spring framework and it is equal to DriverMangerClass.It means Spring framework internally opens a new connection.
- BasicDataSource is given by apache .**common-dbcp** project and it is better than DriverManagerDataSource.Because BasicDataSource has inbuilt connection pooling implementation.

4. Configuration

In Spring Configuration file, we need to configure the following four properties to obtain the connection with the database.

i. driverClassName
ii. url
iii. username
iv. password

Syntax:

1. The following code snippet for *DriverManagerDataSource* configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                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="id" 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="url" value="system"/>
    <property name="password" value="bvrice"/>

 </bean>

</beans>
```

Code Snippet: 1.1 *The above snippet for configure DriverManagerDataSource*

2. The following code snippet for **BasicDataSource**configuration in Spring configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                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="id" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
    <property name="url" value="system"/>
    <property name="password" value="bvrice"/>

 </bean>

</beans>
```

Code Snippet: 1.2 *The above snippet for configure BasicDataSource*

5. Installation:

The following jar file is set as class path for working with Spring Jdbc framework.

1. *org.springframework.jdbc-3.0.6.RELEASE.jar*

6. Central Classes in Spring JDBC framework

6. 1 JdbcTemplate:

- It simplifies the use of JDBC since it handles the creation and release of resources. This helps to avoid common errors such as forgetting to always close the connection.
- It executes the core JDBC workflow like statement creation and execution, leaving Data access using JDBC Spring Framework application code to provide SQL and extract results. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values.
- It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.
- Code using the JdbcTemplate only need to implement callback interfaces, giving them a clearly defined contract. The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The same is true for the CallableStatementCreator interface which creates callable statement. The RowCallbackHandler interface extracts values from each row of a ResultSet.
- The JdbcTemplate can be used within a DAO implementation via direct instantiation with a DataSource reference, or be configured in a Spring IOC container and given to DAOs as a bean reference.

6.1.1 Instantiation:

- ***JdbcTemplate*** class depends on DataSource object, because ***JdbcTemplate*** opens a connection internally with a Database through DataSource Object only.
- ***JdbcTemplate*** class can be created in the following two ways.

    1. JdbcTemplate jtemplate= new JdbcTemplate();
    2. JdbcTemplate jtemplate = new JdbcTemplate(DataSource ds)

- In case first case, DataSource object is injected to ***JdbcTemplate*** by calling setter injection

  Ex: *jt.setDataSource (DataSource obj)*

- Spring IOC Container will inject DataSource object into ***JdbcTemplate***. But we need to configure them into Spring configuration file.

6.1.2 Configuration:

The following code snippet demonstrate how to configure JdbcTemplate in Spring configuration file by using either using code snippet 1.1

Approach 1: Using Setter Injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                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="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
    <property name="url" value="system"/>
    <property name="password" value="bvrice"/>

 </bean>

 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
   <property name="dataSource" ref="dataSource"/>
 </bean>

</beans>
```

Code snippet: 1.3 The above code snippet for JdbcTemplate for instantiation by using setter Injection

Approach 2: Using Constructor Injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                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="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
   <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
   <property name="url" value="system"/>
   <property name="password" value="bvrice"/>

</bean>

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

</beans>
```

Code snippet: 1.4 The above code snippet for JdbcTemplate for instantiation by using constructor Injection

6.1.4 API Methods:

6.1.4.1. void execute(String sql)

- This is method is used to execute both DDL and DML operations on the Database.
- This method allows only static sql command. It means the sql command should not contain '?' symbol
- This is suitable to perform DDL operations on the Database. Because, for DML operations, this method doesn't return the count value back to the programmer.

6.1.4.2 int update(String sql)

- This method is used to execute only DML operations on the database. It means either insert or update or delete.

- If we use a single parameter then we need to pass static SQL command and otherwise we can use the following method.

  Syntax: int update(Strinq sql,object obj[])

- In case of dynamic command, first we need to store all objects into an object array and then we need to pass that object array as a parameter.

  Syntax: *object params[] = {"Spring","jdbc"};*

  int k = jt.update("insert into Spring values(?,?), params);

6.1.4.3 int queryForInt(String sql)

And

int queryForInt(String sql, Object params [])

- This method is used for executing a select operation on database, which returns an integer.
- We can pass either static or dynamic sql command to this method

Syntax:

Static: *int k = jt.queryForInt ("select count (*) from emp");*

6.1.4.4 long queryForLong(String sql)

And

long queryForLong (String sql, Object obj[])

- If our query command (select) is going to return a long value then we use this command.
- We can pass both static and dynamic sql commands to this method.

Syntax:

1. *long l = jt.queryForLong("select sum (sal) from emp");*

2. *long l = jt.queryForLong ("select sum(sal) from emp where deptno = ?",*

new Object []{new Integer (20)});

6.1.4.5 Object queryForObject (String sql, Class clazz)

- This method is used to get the result of select command in the form of required object.
- We have to pass the required class type, in the form a class Object as a second parameter

Syntax:

1. *Object obj = jt.queryForObject("select sysdate from dual" ,Date.class);*
2. *Object obj = jt.queryForObject("select avg(sal) from emp",Integer.class);*

6.1.4.6 List queryForList(String sql)

And

List queryForList (String sql, Object obj [])

- This method is used for selecting one or more rows/records from the database table.
- Internally JdbcTemplate stores all the rows into a array for each row and finally stores all these object array into a collection of type ArrayList and the list object is given back to the programmer.

- While iterating the collection the programmer has type cast the result in to an object array.

Syntax:

```
List l = jt.queryForList ("select * from EMP");
Iterator it = l.iterator();
while (it.hasNext ())
{
Map map = (Map) it.next ();
}
```

Programs from class room.

Loading DataSourse Properties from ResourceBundle

- While configuring a DataSource implementation class into a Spring configuration xml file, we are placing directly the connection properties into the xml file.
- Instead of directly placing the values into the xml file, we can load the values at runtime for the DataSource properties from ResourceBundle.
- If we want to get the DataSource properties at run time from a ResourceBundle then while configuring the bean into xml file. We should put the bundle keys with expression language into the xml file.

Syntax:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="driverClassName" value="${jdbc.driver}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="url" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>

</bean>
```

- In to the above bean definition, the values are passed at runtime from the ResourceBundle.The Bundle is like the following.

```
jdbc.driver = oracle.jdbc.driver.OracleDriver
jdbc.url = jdbc:oracle:thin@localhost:1521:sudheer
jdbc.username = system
jdbc.password= password-1
```

- In Spring Framework, we have a predefined class given called *PropertyPlaceholderConfigure* and this class will load/read the data from bundle and it will write the values into the bean definition of xml
- *PropertyPlaceholderConfigure* is a class given in *org.springframework.beans.factory.config* package.

Programs from class room.

NamedParameterJdbcTemplate

- The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters (as opposed to programming JDBC statements using only classic placeholder ('?') arguments.

- NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work.
- This section will describe only those areas of the `NamedParameterJdbcTemplate` class that Data access using JDBC differ from the `JdbcTemplate` itself; namely, programming JDBC statements using named parameters.

Syntax:

```
package com.cts.spring.controller;

public class NamedJdbcTemplateDemo {

        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public void setDataSource(DataSource dataSource) {
                this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(
                                dataSource);
        }

        public int countOfActorsByFirstName(String firstName) {
                String sql = "select count(0) from T_ACTOR where first_name = :first_name";
                SqlParameterSource namedParameters = new MapSqlParameterSource(
                                "first_name", firstName);
                return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
        }
```

- If you like, you can also pass along named parameters (and their corresponding values) to a NamedParameterJdbcTemplate instance using the (perhaps more familiar) Map-based style. (The rest of the methods exposed by the NamedParameterJdbcOperations - and implemented by the NamedParameterJdbcTemplate class) follow a similar pattern and will not be covered here.)

```
package com.cts.spring.controller;

public class NamedJdbcTemplateDemo {

        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public void setDataSource(DataSource dataSource) {
                this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(
                                dataSource);
        }

        public int countOfActorsByFirstName(String firstName) {
                String sql = "select count(0) from T_ACTOR where first_name = :first_name";
                Map namedParameters = Collections.singletonMap("first_name", firstName);
                return this.namedParameterJdbcTemplate
                                .queryForInt(sql, namedParameters);
        }
```

- Another nice feature related to the `NamedParameterJdbcTemplate` (and existing in the same Java package) is the `SqlParameterSource` interface. You have already seen an example of an implementation of this interface in one of the preceding code snippets (the `MapSqlParameterSource` class).
- The entire point of the `SqlParameterSource` is to serve as a source of named parameter values to a `NamedParameterJdbcTemplate`.The `MapSqlParameterSource` class is a very simple implementation, that is simply an adapter around a `java.util.Map`, where the keys are the parameter names and the values are the parameter values.
- Another `SqlParameterSource` implementation is the `BeanPropertySqlParameterSource` class. This class wraps an arbitrary JavaBean (that is, an instance of a class that adheres to the JavaBean conventions), and uses the properties of the wrapped JavaBean as the source of named parameter values.

  Syntax:

```
package com.cts.spring.controller;

public class Actor {
        private Long id;
        private String firstName;
        private String lastName;

        public String getFirstName() {
                return this.firstName;
        }

        public String getLastName() {
                return this.lastName;
        }
}
```

```
package com.cts.spring.controller;

public class NamedJdbcDemo {
        // some JDBC-backed DAO class...
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public void setDataSource(DataSource dataSource) {
                this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(
                                dataSource);
        }

        public int countOfActors(Actor exampleActor) {
                // notice how the named parameters match the properties of the above
                // 'Actor' class
                String sql = "select count(0) from T_ACTOR where first_name = :firstName and
last_name = :lastName";
                SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(
                                exampleActor);
                return this.namedParameterJdbcTemplate
                                .queryForInt(sql, namedParameters);
        }

}
```

- Remember that the NamedParameterJdbcTemplate class wraps a classic JdbcTemplate template; if you need access to the wrapped JdbcTemplate instance (to access some of the functionality only present in the JdbcTemplate class), then you can use the getJdbcOperations () method to access the wrapped JdbcTemplate via the JdbcOperations interface.

Calling Procedures/Functions:

- To call a procedure or a function from a database, we- have two approaches in Spring i.e. by extending StoredProcedure class and by without extending it.

- If we want to call a procedure or a function then the simple approach is by extending our bean class from StoredProcedure. (Abstract class but not abstract methods).

- If we want to call procedures/functions of the database then the following steps are need to be followed.

    1. Extend the bean class from StoredProcedure.

    2. Call the base class constructor from our bean class by passing datasource object, name of the procedure name or function as parameters.

       Syntax:

       super (datasource,"pro1");

    3. Declare the keys in the form of sqlParameter objects that are used as keys for map objects.

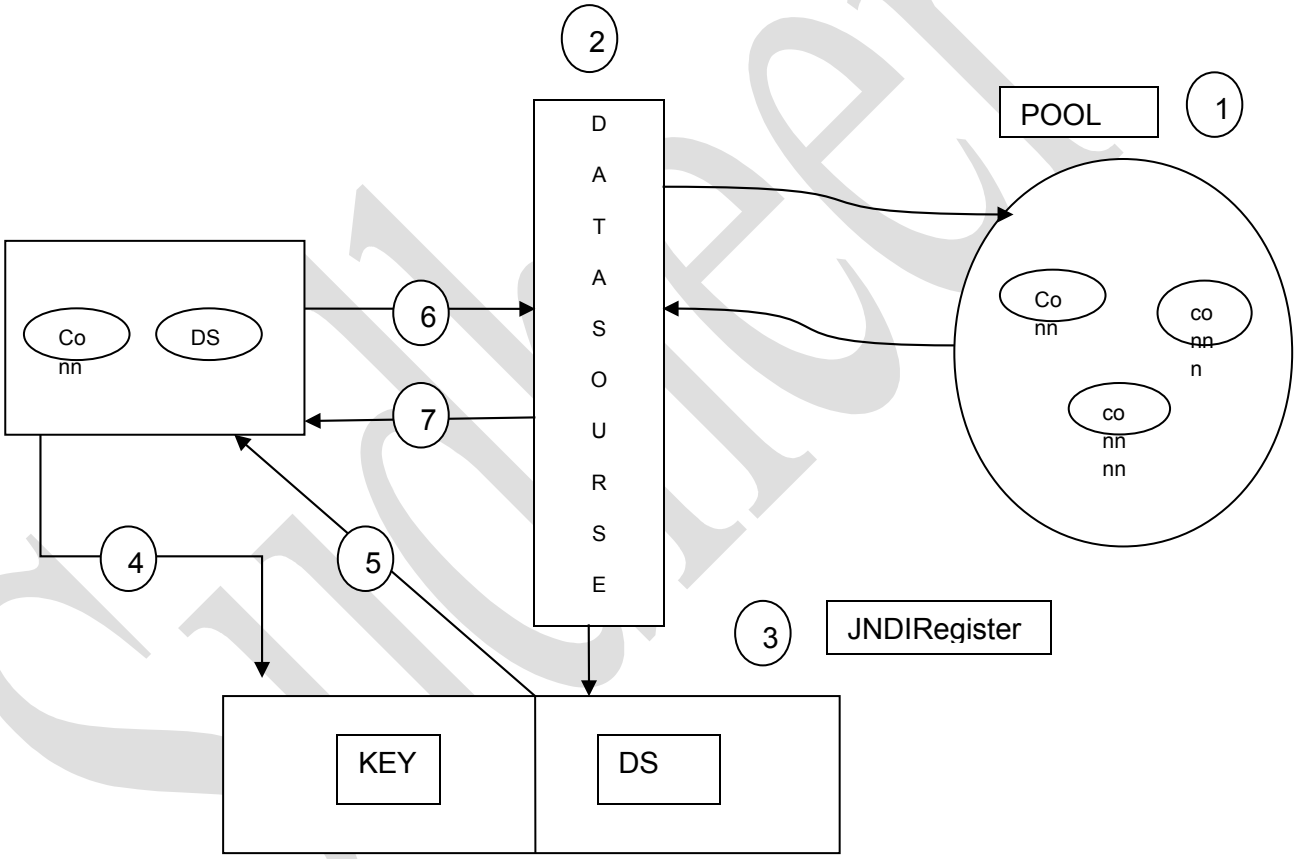
    4. Call execute () method from the business method of the bean by passing the input values in the form of map object.

    5. Read the output values returned into the map object from the procedure or function.

Example sees in class room.

Using Connection Pooling in Spring:

- When we configure BasicDataSource into Spring configuration xml file then implicitly Spring application is using connection pool service. Because BasicDataSource class has a inbuilt connection pool service.
- The inbuilt connection is only suitable for small scale applications, but not for production level applications.
- If a Spring application or any other java application wants to get a connection from an external connection pool then the application need a mediator object called DataSource, which is configured in the server.
- When a DataSource is configured by the server administrator then it will be stored in JNDIRegistery with some JNDI name.
- If any Java application wants to get connection pool service of a server then the application has to get the DataSource object from the Registery.
- If an application wants to get DataSource object then it need JNDI name.
- In case of Spring framework, the


2
POOL
1
D
A
T
A
S
O
U
R
S
E
Co
nn
DS
6
7
Co
nn
co
nn
n
co
nn
nn
4
5
3
JNDIRegister
KEY
DS

Spring AOP

Introduction

- We are aware that nowadays the most widely used methodology for developing enterprise applications is Object oriented programming (OOP).The OOP methodology was introduced in the 1960s to reduce the complexity of the software and improve its quality. This achieved by modularization and reusability. This allow developers to view the Software as a collection of objects that interact with each other, allowing easy solving of critical problems and developing reusable units of software
- While implementing business logic for the real time applications we need not only the business logic but also some services to it, to make it as enterprise logic.
- While implementing the business logic in the business methods we can combinable implement both the logic and its required services into the business methods.
- In order spring framework the services that are overlapping on the business logic are called as ‘cross-cutting’ concerns or ‘cross-cutting’ functionalities.

- **Example**

```
public class Account {

	public void withDraw()
	{
		// Authentication or Logging
or Transaction
		// Business Logic
	}
	public void deposit()
	{
		// Authentication or Logging
or Transaction
		// Business Logic
	}

}
```

- If combinable implement the business logic and the required services then there are some drawbacks

  a. The services and the business logic in each method, increases the size of a Java Class. Therefore complexity increases.

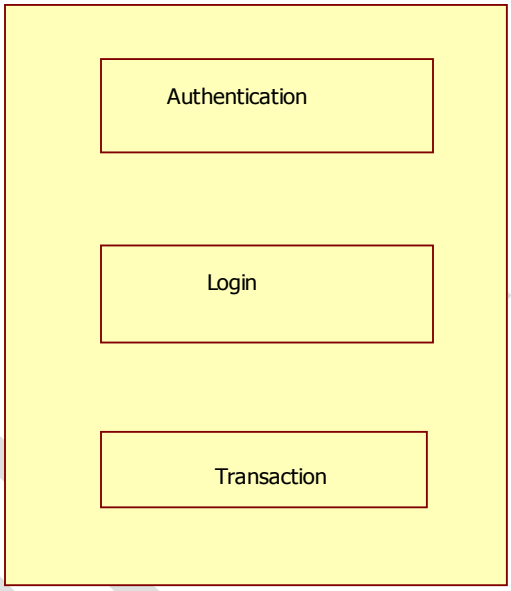
  b. If the same services are required in another business method then we cannot reuse those services and in each method separate implementations are required.

  c. If any changes in some service in one method then that changes does not impact on same services of another method.

  d. Selecting services dynamically at runtime is not possible.

- In order overcome the above problems we need to separate the business logic and the service. We can call this separation of the business logic and cross-cutting functionality as aspect oriented programming.

```
public class Account {

	public void withDraw()
	{
		// Business Logic
	}
	public void deposit()
	{
		// Business Logic
	}

}
```



- The AOP the business logic and cross cutting functionality are implemented separately and executed at runtime as combinable.

Aspect Oriented Programming (AOP)

- Aspect Oriented Programming (AOP) is a programming methodology that allows the developers to build the core concerns without making them aware of secondary requirements by introducing aspects that encapsulate all the secondary requirements logic and the point of execution where they have to be applied.

Terminology

- **Aspect:**

  a. Aspect represents or denotes a cross-cutting functionality.

  b. One real time service is required for a business logic called one aspect.

  c. An aspect denotes only the cross cutting functionality but not its implementation.

- **Advice:**

  a. Advice defines what needs to be applied and when

  b. The implementation of an aspect is called as an Advice

  c. An Advice provides the code for implementation of service.

- **JoinPoint**

  a. JoinPoint is where Advice is applied.

  b. While executing the business logic of a business method, the additional services are needed to be injected at different places/points we call such points as JoinPoint.

  c. At a JoinPoint, a new service will be added into the normal flow of a business method.

d. In the execution of a business method, the services are required at the following three places. We call them as the three Join Points.

i. Before Business logic of a method states

ii. After Business logic of a method is completed.

iii. If the business logic throws an exception at runtime.

- **Pointcut**

a. Pointcut is the combinations of different Joinpoints where the advices need to be applied. (or) A Pointcut defines what advices are required at what Joinpoints.

b. All business methods of a class does not required all services.

- **Introduction**

a. An Introduction adds new properties and new methods to already existing classes at runtime by without making any changes to the class.

b. An Introduction introduces new behavior and a new state to the existing class at runtime.

- **Target**

a. A target is a class which is going to be advised a target class is nothing but a class which is not yet added with AOP feature. It means a target is 'PRE AOP OBJECT'.

- **Proxy**

  a. A proxy denotes an object that is applied with AOP feature. An object at PRE-AOP called as target and a object at POST-AOP stage is called as a Proxy.

- **Weaving**

  a. Weaving is the process of applying advices to a target object to get a new proxy object.

  b. Though weaving process a target object will converted to proxy object, by adding advices.

  c. Weaving Process Types

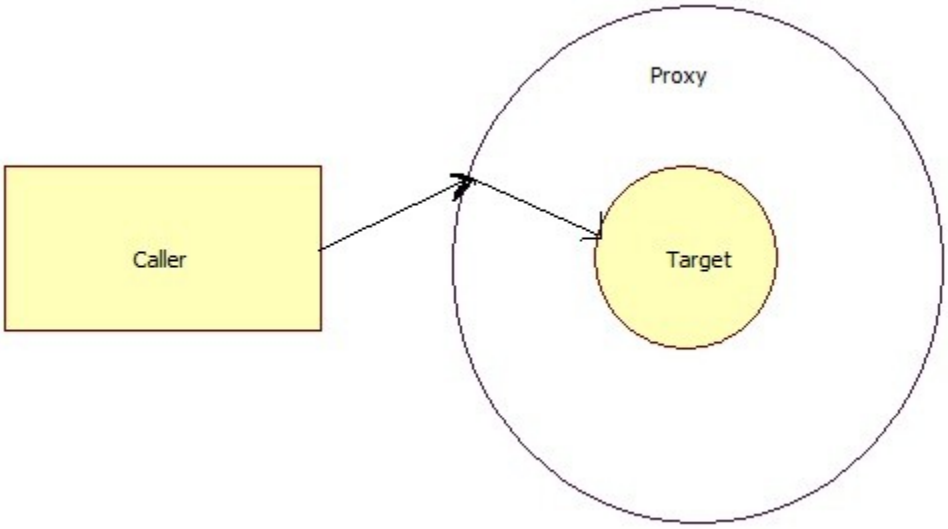
  i. **Compile Time weaving** is a process of injecting byte code of advice is applied on JoinPoint at compile time.

  ii. **Class Loader Weaving** is a process of injecting byte code of advice is applied on JoinPoint a Class Loading time.

  iii. Runtime weaving (Spring way)

  In this, target bean will be shielded with proxy bean and created by spring framework. Whenever Caller invokes methods on target bean then spring frameworks invokes proxy bean and then understand what advices methods need to be applied on target bean and executes real method of

target bean along with their advices.



- **Advisor (or) Pointcut Advisor:**

  An advisor advising advices to various Joinpoints we call advisor has a pointcut advisor. We have two types of Advisor.

Advice API:

- Advice is an object that includes API invocation to systemwide concerns representing the action to perform at a joinpoint specified by pointcut.

- The different types of advices that Spring AOP framework supports are:

    i. **Before advice:** The advice that executes before the joinpoint.

    ii. **After returning advice:** The advice that executes after the joinpoint execution completes normal termination.

    iii. **Throws advice:** The advice that executes after the joinpoint execution if it completes with abnormal termination.

    iv. **Around advice:** The advice that can surround the joinpoint providing the advice before and after executing the joinpoint even is controlling the joinpoint invocation.

Before Advice

- **T**he before advice is the advice that executes before the joinpoint.That means this type of advice allows us to intercept the request and apply the advice before the execution of joinpoint.

- In order to create a BeforAdvice our class should implement MethodBeforeAdvice interface.

- MethodBeforeAdvice is an interface given in ***org.springframework.aop*** package.

- The ***org.springframework.aop.MethodBeforeAdvice*** interface has only one method with following signature.

- **For Example**

```
package com.nit.spring.aop;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

/**
 * @author sudheer
 *
 */
public class WelcomeAdvice implements MethodBeforeAdvice{

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


	}

}
```

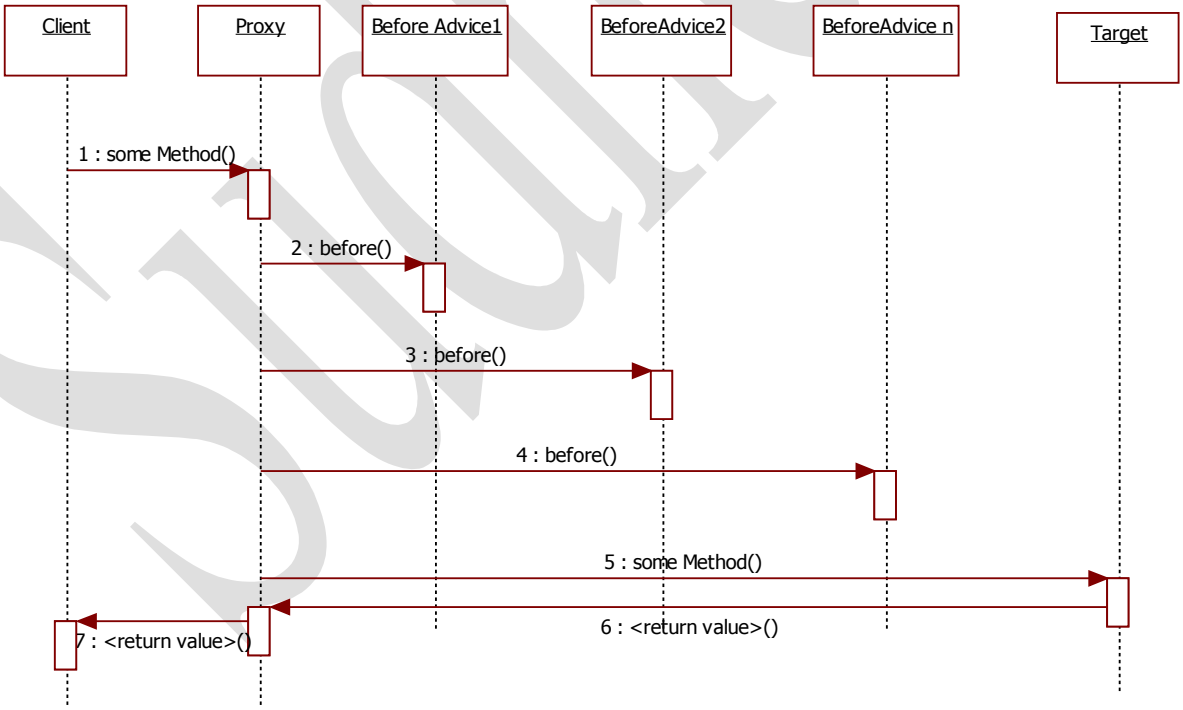
a. The first parameter **Method** is a class of ***java.lang.reflect*** package and this parameter is used to access the name of the business method through ***getName().***

b. The second parameter ***Object [] args*** is used to access the parameters of the business method.The parameters are given to the before() method by the container in the form of an ***Object [] args.***

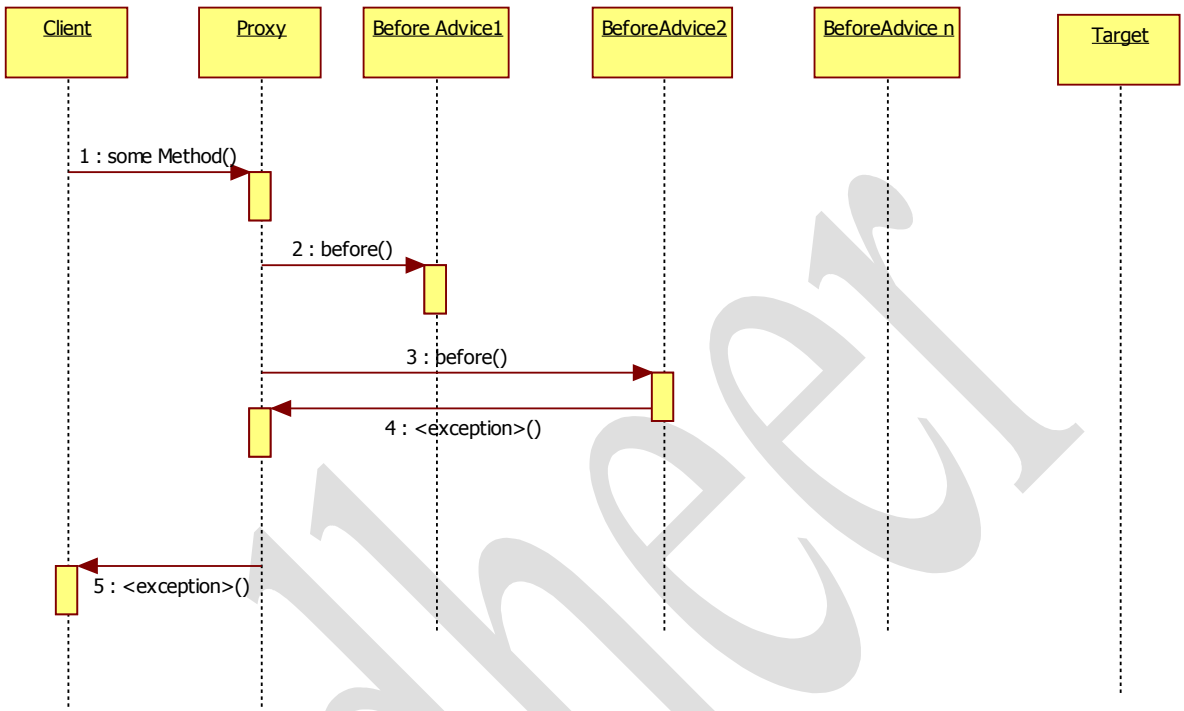
c. The Third parameter is an object to whom this service is going to apply.

- The before() method can encapsulate the custom code that has to execute before the joinpoint executes.
- The following diagram shows a sample sequence diagram explaining the operations when before advice and joinpoint execution terminates normally.Moreover,if this advice decides to discontinue the joinpoint execution then it can throw some exception. But if the before advice throws an exception, apart from terminating the further execution of the interceptor chain, the client is

throw with exception, not a normal termination

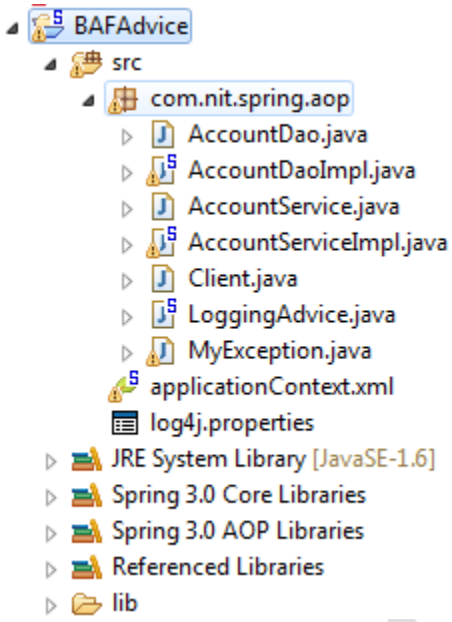


- The following sequence diagram describing the operations when before advice execution terminates abnormally, that is, throwing exception.



- As shown in figure, in the case where before advice throws an exception then an exception is thrown to the client without continuing further the interceptor's execution and target object method.

- The following diagram describing the exception handling done by proxy when advice throws an exception.



b. **AccountService.java**:

This Interface have two abstract methods performing withdraw and deposit methods.

```
package com.nit.spring.aop;

public interface AccountService {

	boolean deposit(int acc,double amt)throws MyException;
	boolean withDraw(int acc,double amt)throws MyException;
}
```

c. **AccountServiceImpl.java**

```
package com.nit.spring.aop;

public class AccountServiceImpl implements AccountService {

	private AccountDao accountDao;

	public AccountServiceImpl(AccountDao accountDao) {
		this.accountDao = accountDao;
	}

	@Override
	public boolean deposit(int acc, double amt) throws MyException {
		accountDao.setBalance(acc,amt);
		return true;
	}

	@Override
	public boolean withDraw(int acc, double amt) throws MyException {
	  double balance = accountDao.getBalance(acc) - amt;
		return true;
	}
```

d. **AccountDao.java**

```
package com.nit.spring.aop;

public interface AccountDao {

	void setBalance(int accno,double amt) throws MyException;

    double getBalance(int accno) throws MyException;

}
```

e. **AccountDaoImpl.java**

```
package com.nit.spring.aop;

import org.apache.log4j.Logger;

public class AccountDaoImpl implements AccountDao{

	private static Logger log = Logger.getLogger(AccountDaoImpl.class);
	@Override
	public double getBalance(int accno) throws MyException {
		log.info(" Inside getBalance with accno "+accno);
		return 500;
	}

	@Override
	public void setBalance(int accno, double amt) throws MyException {
		log.info("Inside setDeposit method with "+accno);
		double totalamt = getBalance(accno) + amt;
		System.out.println(" Total Balance "+amt);
	}

}
```

f. MyException.java

```
package com.nit.spring.aop;

public class MyException extends Exception {

	public MyException() {

	}
	public MyException(String message)
	{
		super(message);
	}
}
```

g. LoggingAdvice.java

```
package com.nit.spring.aop;

import java.lang.reflect.Method;

import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAdvice implements MethodBeforeAdvice {

	@Override
	public void before(Method method, Object[] args, Object target)
			throws Throwable {
		Logger log = Logger.getLogger(target.getClass());
		log.info(" Method Name " + method.getName() +"  Method
Parameters "+args);

	}

}
```

h. log4j.properties

```
log4j.rootLogger = info,myapp
log4j.appender.myapp = org.apache.log4j.FileAppender
log4j.appender.myapp.file = D://sudheer.html
log4j.appender.myapp.layout=org.apache.log4j.HTMLLayou
t
log4j.appender.myapp.append =false
```

i. applicationContext.xml

```
package com.nit.spring.aop;

import java.lang.reflect.Method;

import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAdvice implements MethodBeforeAdvice {

	@Override
	public void before(Method method, Object[] args, Object target)
			throws Throwable {
		Logger log = Logger.getLogger(target.getClass());
		log.info(" Method Name " + method.getName() +"  Method
Parameters "+args);

	}

}
```

h. log4j.properties

```
log4j.rootLogger = info,myapp
log4j.appender.myapp = org.apache.log4j.FileAppender
log4j.appender.myapp.file = D://sudheer.html
log4j.appender.myapp.layout=org.apache.log4j.HTMLLayou
t
log4j.appender.myapp.append =false
```

i. applicationContext.xml

j.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

      <bean id="accountDaoImpl" class="com.nit.spring.aop.AccountDaoImpl" />

      <bean id="accountServiceImpl"
class="com.nit.spring.aop.AccountServiceImpl">
            <constructor-arg>
                  <ref bean="accountDaoImpl" />
            </constructor-arg>
      </bean>

      <bean id="loggingAdvice" class="com.nit.spring.aop.LoggingAdvice" />



      <bean id="proxyBean"
class="org.springframework.aop.framework.ProxyFactoryBean">
            <property name="proxyInterfaces">
                  <value>com.nit.spring.aop.AccountService</value>
            </property>

            <property name="target">
                  <ref bean="accountServiceImpl" />
            </property>

            <property name="interceptorNames">
                  <list>
                        <value>loggingAdvice</value>

                  </list>
            </property>

      </bean>

</beans>
```

k. Client.java

```
package com.nit.spring.aop;

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

public class Client {

	/**
	 * @param args
	 * @throws MyException
	 */
	public static void main(String[] args) throws MyException {

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

		AccountService acc = (AccountService)context.getBean
("proxyBean");

		acc.withDraw(1,200.00);

	}

}
```

Output:

sudheer.html

After Returning Advice

- The after returning advice is the advice that executes after the joinpoints invocation completes with normal termination.
- This advice has to be a subtype of ***org.springframework.aop.AfterReturningAdvice*** interface.

- The ***org.springframework.aop.AfterReturningAdvice*** interface has only one abstract method.

- **Syntax**

```
package com.nit.spring.aop;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class GoodByeAdvice implements AfterReturningAdvice {

	@Override
	public void afterReturning(Object retValue, Method method, Object[] args,
			Object target) throws Throwable {
		//Business Logic
	}

}
```

- Where the first parameter contains return value of the business method or null.

- If a business method return type is void then the return value contains null

- In this afterReturning advice, we can access the return value. But we can not modify this return value.

- **Example: see class room example**

Throws Advice:

- The throws advice is the advice that executes after the joinpoints invocation completes with an abnormal termination.
- This advice has to be a subtype of ***org.springframework.aop.ThrowsAdvice*** interface.
- The ***org.springframework.aop.ThrowsAdvice*** is a marker interface and does not declare any methods.

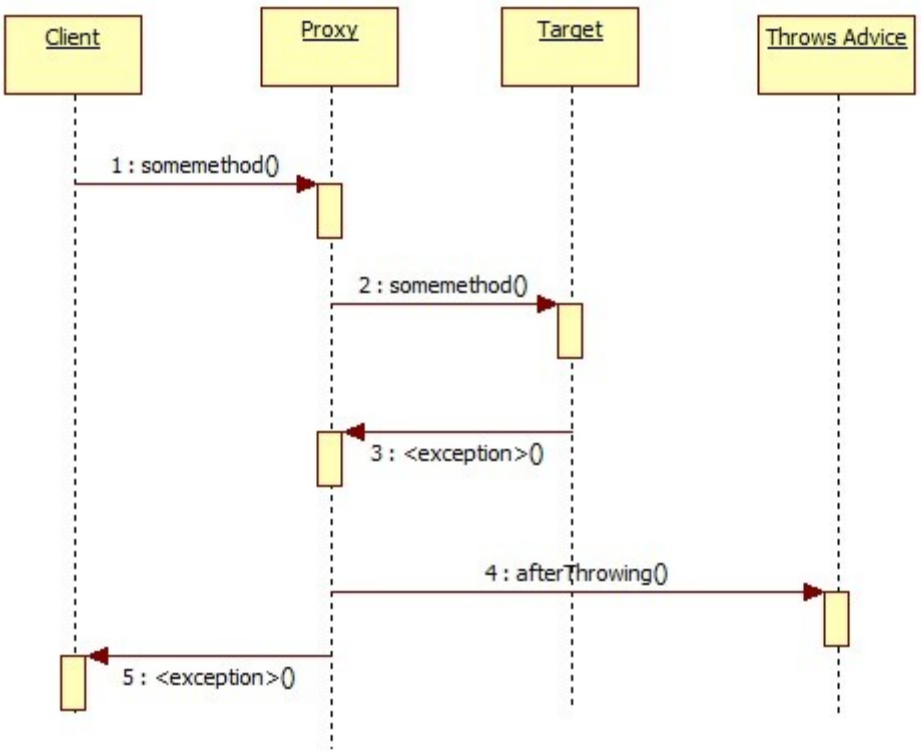
- This type of advice should implement one or more methods with the following method signature.
- The throws advice can listen for the exceptions thrown by the method invoked on the target object but not the exceptions thrown by the advices configured for the target object.
- The throws advice can handle the exception but it cannot stop the exeception throwing to the client.
- **Syntax**

-

```
public class ThrowsAdvice implements org.springframework.aop.ThrowsAdvice {

	public void afterThrowing(ArithmeticException ae)
	{
		//Business Logic
	}

	public void afterThrowing(Method method,Object args[],Object target)
	{
		//Business Logic
	}
}
```

- The afterThrowing() method can encapsulate the exception handling logic like in an instance when an ArthimeticException is thrown we want to prepare a relevant message and send an email to the administrator.

- **SequenceDiagram**



- **Example– see class room example**

Around Advice:

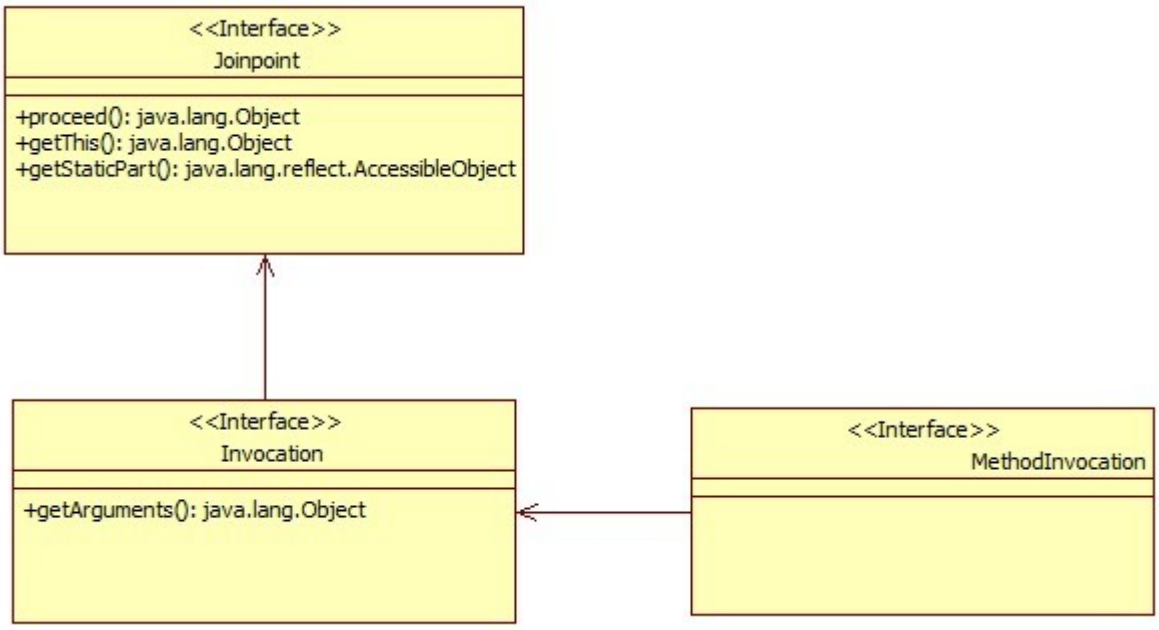
- Around advice is the combination of both before and After advices.
- In a single advice it is possible to implement both before and after services.
- Around advice is not given by Spring AOP and it is an opensource implementation call AOP Alliance.
- Around Advice can be used by any framework which is supporting AOP.
- To create around advice, our class should implement an interface called ***MethodInterceptor.***
- In Around Advice, we implement before and after services in a single method called invoke().In order to separate before and after services and to execute business logic of the method in the middle, we call proceed().
- **Syntax**

```
public class TimerAdvice implements MethodInterceptor{

	@Override
	public Object invoke(MethodInvocation mi) throws Throwable
{

		// Before Service

	    Object o = mi.proceed();
		return o;

		//AfterService

	}

}
```

- The ***org.aopalliance.intercept.MethodInvocation*** allows us to get the details of the invoked method of the target object, method arguments, and Joinpoint.

- The MethodInvocation interface extends invocation, which is a subtype of joinpoint.

- **Class Diagram**



- The invoke() method implemented in around advice can call the proceed() method on the given MethodInvocation object.

- The code that has to execute before the joinpoint execution has to be written before calling proceed() method and the code to execute after the joinpoint execution has to be written after the procedd() method.

- **Sequence Diagram:**