



Spring Framework 3.0

(version 3.0)



Document History



Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
21-Jul-2006	Ver1.0	1.2.6	Shrilata Tavargeri	NA
Jan-2009	Ver2.0	2.5	Shrilata Tavargeri	Changed material with new changes introduced in ver 2.5 and inputs from BU
Aug-2011	Ver 3.0	3.0	Shrilata Tavargeri	Changed material with new changes introduced in ver 3.0 and inputs from BU
June 2013	Ver 4.0	3.0	Mohan Chinnaiah	Revamped materials according to new requirements



Keep this as a hidden slide.

Note to co-ordinators: Not to be printed for the class book.

Course Goals and Non Goals



➤ Course Goals

- Understand the benefits of using Spring
- Understand the principles of IoC and AOP
- Be able to use AOP to handle cross-cutting concerns
- Connect business objects to persistent stores using Spring's DAO modules
- Use the Spring MVC web framework to develop flexible web applications
- Introduction to Spring Testing



➤ Course Non Goals

- Design patterns, Spring Integration with different technologies



- 3 -

Pre-requisites



- Core Java , Java 5 features and JDBC
- XML, DBMS/SQL
- Servlets, JSP
- Concepts of MVC, Design patterns



+ 4 +

Intended Audience



- All Java application developers especially Enterprise Java Programmers
- Software designers



© SINGH TECHNOLOGIES, INC.
IGATE.COM



- 5 -

Day Wise Schedule



➤ Day 1

Lesson 1: Introduction to Spring Framework, IoC

➤ Day 2

Lesson 2 : SpEL (Spring Expression Language)

Lesson 3: DAO support and JDBC in Spring

➤ Day 3

Lesson 4: Spring MVC framework

➤ Day 4

Lesson 5: AOP (Aspect Oriented Programming)

Lesson 6: Introduction to Spring Testing



- 6 -

Table of Contents



1. Lesson 1: Introduction to Spring Framework

- 1.1. What is Spring Framework, Benefits of Spring
- 1.2. The Spring architecture
- 1.3. IOC – Inversion of control, wiring beans
- 1.4. Bean containers, lifecycle of beans in containers
- 1.5. Customizing beans with PostProcessors
- 1.6. Annotation-based configuration

2. Lesson 2: Introduction to SpEL (Spring Expression Language)

- 2.1. SpEL Expression fundamentals
- 2.2. Expression Language features
- 2.3. Reduce configuration with @Value

- 7 -



Table of Contents



3. Lesson 3: DAO support in Spring

3.1. Data access using JDBC, JdbcTemplate

4. Lesson 4: Spring MVC framework

4.1. Introduction: DispatcherServlet, Handler mappings, Resolving views

4.2. Annotation-based controller configuration

5. Lesson 5: AOP (Aspect Oriented Programming)

5.1. AOP concepts

5.2. AOP support in Spring using @AspectJ support

5.3. AOP support in Spring using Schema-based AOP support

6. Lesson 6: Introduction to Spring Testing

6.1. Testing Spring applications

6.2. Spring's Test Context framework

- 8 -



References



- **Spring in Action, Third Edition, Manning publications by Craig Walls**
- **Spring-framework-reference.pdf from SpringSource (this is available in the downloaded Spring software)**



- 9 -

Software required



- **JDK version 1.5 + with help, Netscape or IE**
- **MS-Access/Connectivity to Oracle database**
- **JBoss AS 4.2 +**
- **Eclipse 3.5+**
- **Spring 3.0 API with docs**

- 10 -



Other Parallel Technology Areas

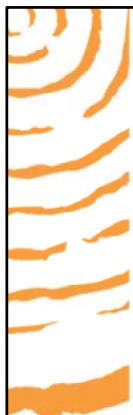


- **EJB 3.0**
- **PicoContainer**
- **NanoContainer**
- **Keel Framework**
- **Google Guice**

- 11 -



- PicoContainer: is an exceptionally small DI (Dependency Injection) container that allows to use DI for your application without introducing any dependencies other than PicoContainer itself
- NanoContainer: is an extension to PicoContainer for managing trees of individual PicoContainer containers.
- Keel Framework: is more of a metaframework, in that most of its abilities come from other frameworks that are all brought together under one roof.
- Google Guice: focuses purely on DI.



Spring Framework 3.0

Lesson 1: Spring Basics and IoC



Lesson Objectives



- **Introduction to Spring Framework**
 - Learn about the Spring Framework, its benefits and architecture
 - Learn about the IoC (Inversion of control) and how it allows wiring beans
 - Learns the types of bean factories and life-cycle of beans in these factories
 - Understand how to apply Annotations to Spring applications
- **Injecting dependencies through setter and constructor injections**
- **Wiring Beans**
- **Bean containers**
 - Life cycle of Beans in the factory container
 - BeanPostProcessors and BeanFactoryPostProcessors
- **Annotation-based configuration**



- 2 -



Introduction

- **December 1996 – JavaBeans makes its appearance.**
 - Intended as a general-purpose means of defining reusable application components
 - Used more as a model for building user interface widgets
- **Sophisticated applications often require services not directly provided by the JavaBeans specification**
- **March 1998 – EJB was published.**
 - But EJBs are complicated in a different way, that is, they mandate deployment descriptors and plumbing code

It all started with a bean. In 1996, the Java programming language was still a young, exciting, up-coming platform. Many developers flocked to the language because they had seen how to create rich and dynamic web applications using applets. But they soon learned that there is more to this strange new language than juggling animated cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but stayed for the components.

It was in December of that year that Sun Microsystems published the Java-Beans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components, they have been primarily used as a model for building user interface widgets. They seemed too simple to be capable of any "real" work; enterprise developers wanted more.

Sophisticated applications often require services that are not directly provided by the JavaBeans specification such as transaction support, security, and distributed computing. Therefore in March 1998, Sun published the 1.0 version of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing the much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. In fact, except in name, EJB bears very little resemblance to the original Javabeans specification.

1.1 : Introduction to Spring Framework

Introduction

 iGATE
ITOPS for Business Outcomes

- **Many successful applications were built based on EJB**
 - But EJB never really achieved its intended purpose, which is to simplify enterprise application development
- **Java development comes full circle**
 - New programming techniques like including aspect-oriented programming (AOP) and inversion of control (IoC) are giving JavaBeans much of the power of EJB



Despite the fact that many successful applications have been built based on EJB, it never really achieved its intended purpose: to simplify enterprise application development. It is true that EJB's declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. But EJBs are complicated in a different way by mandating deployment descriptors and plumbing code (home and remote/local interfaces). As a result, its popularity has started to wane in recent years, leaving many developers looking for an easier way.

Now Java development is coming full circle. New programming techniques, including aspect-oriented programming (AOP) and inversion of control (IoC), are giving JavaBeans much of the power of EJB. These techniques furnish JavaBeans with a declarative programming model reminiscent of EJB, but without all of EJB's complexity. No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.

And that's where Spring steps into the picture.

In all fairness, the latest EJB specification (EJB 3) has evolved to promote POJO-based programming model and is simpler than its predecessors.

Quick history :

Spring Framework project founded in Feb 2003
Release 1.0 in Mar 2004
Release 1.2 in May 2005
Release 2.0 in Oct 2006
Release 2.5 in Nov 2007
Release 3.0 in Dec 2009

1.1 : Introduction to Spring Framework

What is Spring?



- Spring is an open source framework created by Rod Johnson, Juergen Hoeller et all
- Addresses the complexity of enterprise application development
- Any java application can benefit from Spring in terms of simplicity, testability and loose coupling

- 5 -



Spring makes it easy to use POJO (Plain Old Java Objects) to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't restricted to server-side development. Any java application can benefit from Spring in terms of simplicity, testability and loose coupling.

Put simply, Spring is a lightweight inversion of control and aspect-oriented container framework. Breaking this description down makes it simpler:

Lightweight: Spring is Lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a single jar file of 2.5 MB approximately. Processing overhead required by Spring is negligible. Moreover, Spring is non-intrusive; objects in a Spring-enabled application typically have no dependencies on Spring-specific classes.

Inversion of control: Spring promotes loose coupling through a technique known as inversion of control (IoC) or also known popularly as Dependency Injection (DI). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. IoC can be thought of as JNDI in reverse – instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

Aspect-oriented : Spring comes with rich support for aspect-oriented programming that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they are supposed to do – perform business logic – and nothing more. They are not responsible for other system concerns such as logging or transactional support.

Container: Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. You can configure how each of your beans should be created – either create one single instance of your bean or produce a new instance every time one is needed based on a configurable prototype – and how they should be associated with each other.

Framework: Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration etc) leaving the development of application logic to user.

All these attributes of Spring enable you to write code that is cleaner, more manageable and easier to test.

1.1 : Introduction to Spring Framework

Why Spring?

- **Spring simplifies Java development**
- **With Spring, complexity of application is proportional to the complexity of the problem being solved**
- **Essence of Spring is to provide enterprise services to POJO.**
- **Spring employs four key strategies:**
 - Lightweight and minimally invasive development with plain old Java objects (POJOs)
 - Loose coupling through dependency injection and interface orientation
 - Declarative programming through aspects and common conventions
 - Boilerplate reduction through aspects and templates

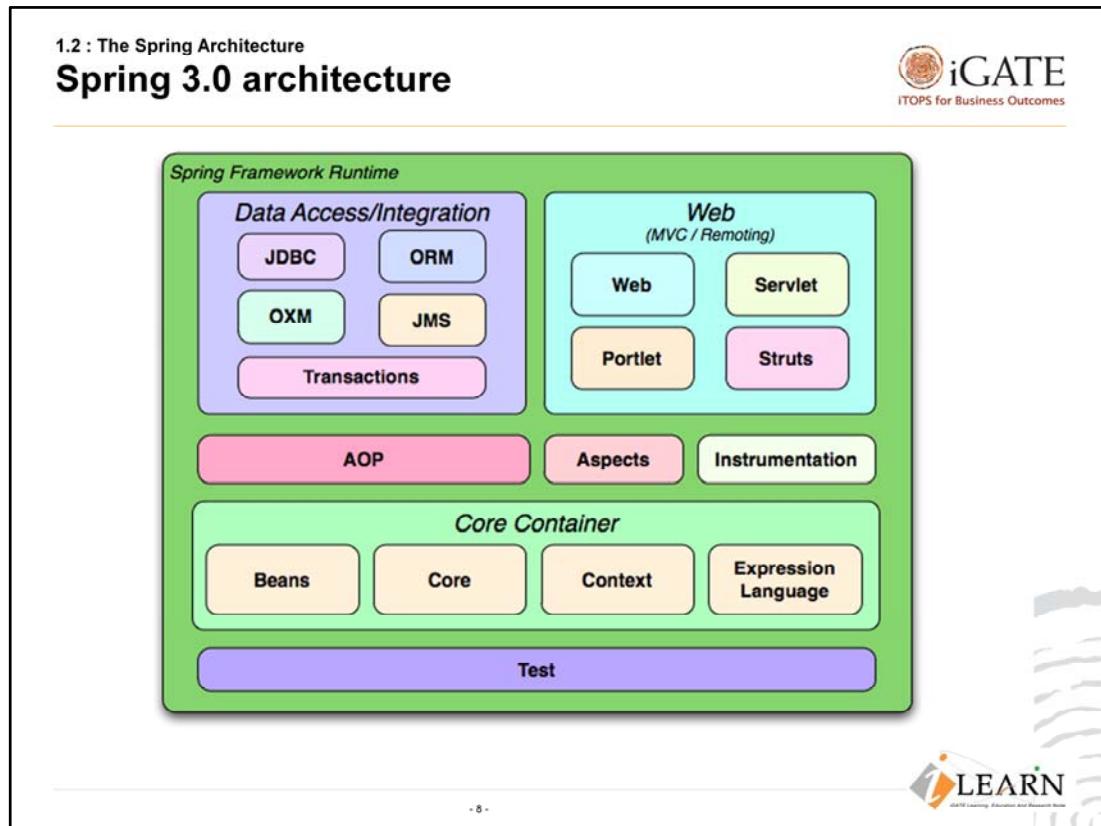
+ 7 +

 LEARN
iGATE Learning, Education and Research Network

To put it simply, Spring makes developing enterprise applications easier. The complexity of your application is proportional to the complexity of the problem being solved.

Some key Spring values are:

1. **It's a non-invasive framework** : Traditional frameworks force application code to be aware of the framework, implementing framework specific interfaces or extending framework-specific classes. Spring aims to minimize the dependence of application code on the framework.
2. **Promotes pluggability** : Spring encourages you to think of application objects as named services. Thus you can swap one service for another without affecting the rest of the application.
3. **Aims to facilitate good programming practices**, such as programming to interfaces: Using an IoC container like Spring reduces the complexity of coding to interfaces, rather than classes.
4. **Spring applications are easy to test** : Application objects will generally be POJOs and POJOs are easy to test; dependence on Spring APIs will normally be in the form of interfaces that are easy to stub or mock.
5. **Does not reinvent the wheel** : Spring does not introduce its own solution in areas such as O/R mapping where there are already good solutions. It also does not implement its own logging abstraction, connection pool, distributed transaction coordinator or other system services that are already well-served in other products or application servers. However Spring does make these existing solutions significantly easier to use.



The Spring Framework is composed of several distinct modules. When you download and unzip the Spring Framework distribution, you'll find 20 different JAR files in the dist directory. These JAR files that make up Spring can be arranged in one of six different categories of functionality - Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test. When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But this modularity also gives you the freedom to choose the modules that suit your application.

Core Container : The Core Container consists of the Core, Beans, Context, and Expression Language modules.

- The *Core and Beans* modules are the most fundamental part of the framework. It defines how beans are created, configured and managed. The BeanFactory (a sophisticated implementation of the factory pattern and a primary component of this module) applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code
- The *Context* module builds on the solid base provided by the *Core and Beans* modules. The core modules bean factory makes Spring a container, but the context modules makes it a framework. The Context module adds support for internationalization, event-propagation, resource-loading etc.
- The *Expression Language* module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration : The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.
- The *ORM* module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, XStream etc.
- The Java Messaging Service (JMS) module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (plain old Java objects)*.

Web : The *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

- Spring's *Web* module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The *Web-Servlet* module contains Spring's model-view-controller (*MVC*) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.
- The *Web-Struts* module contains support classes for integrating a classic Struts web tier within a Spring application. This support is however deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.
- The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

AOP and Instrumentation : AOP (Aspect Oriented Programming) enables behavior that would otherwise be scattered through different methods to be modularized in a single place.

- The separate *Aspects* module provides integration with AspectJ.
- The *Instrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

Test : The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

1.3 : Spring Jumpstart

Spring Jumpstart with HelloWorld

```

    package training.spring;
    public class HelloWorld {
        public void sayHello(){
            System.out.println("Hello Spring 3.0");
        }
    }

    public class HelloWorldClient {
        public static void main(String[] args) {
            XmlBeanFactory beanFactory = new XmlBeanFactory(
                new ClassPathResource("HelloWorld.xml"));
            HelloWorld bean = (HelloWorld) beanFactory.getBean("HWBean");
            bean.sayHello();
        }
    }

```

<?xml>
<beans>
<bean id="HWBean" class =
"training.spring.HelloWorld" />
</beans>

The Spring configuration file

Output:
Hello Spring 3.0

LEARN
iGATE Learning, Education and Research Network

Dependency injection is the most basic thing that Spring does. We shall be covering this in detail later in the session. For now, let us see how Spring works with an example.

Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application.

Difference lies in how these classes are configured and introduced to each other. Typically a Spring application has an XML file that describes how to configure the classes, known as Spring configuration file.

Pls. refer to Appendix-A for a detailed explanation of converting a traditional Java application into a Spring-based application.

Look in slide above for a typical Hello World application using Spring Framework. Detailed explanation for this code is given in subsequent demos.

1.3 : Spring Jumpstart

Injecting dependencies via setter methods



```
public interface CurrencyConverter {  
    public double dollarsToRupees(double dollars);  
}
```

```
public class CurrencyConverterImpl implements CurrencyConverter {  
    private double exchangeRate;  
    public double getExchangeRate() { return exchangeRate; }  
    public void setExchangeRate(double exchangeRate) {  
        this.exchangeRate = exchangeRate; }  
    public double dollarsToRupees(double dollars) {  
        return dollars * exchangeRate;  
    }  
}
```



- 11 -

The example shows a service class whose purpose is to print the value of dollars converted to rupees. The listing above shows `CurrencyConverter.java`, an interface that defines the contract for the service class.

`CurrencyConverterImpl.java` implements the `CurrencyConverter` interface. Although it is not necessary to hide the implementation behind an interface, its highly recommended as a way to separate the implementation from its contract.

The `CurrencyConverterImpl` class has a single property `exchangeRate`. This property is simply a double variable that will hold the exchange rate passed by its setter method. We can also pass value through the constructor (We shall see this in the next example)

1.3 : Spring Jumpstart

Injecting dependencies via setter methods



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="currencyConverter"
          class="training.Spring.CurrencyConverterImpl">
        <property name="exchangeRate" value="44.50" />
    </bean>
</beans>
```

The configuration file
(CurrencyConverter.xml)



Question now is who will make a call to either the constructor or the `setExchangeRate()` method to set the `exchangeRate` property? The Spring configuration file in the above listing tells how to configure the `CurrencyConverter` service. This XML file declares an instance of a `CurrencyConverterImpl` in the Spring container and configures its `exchangeRate` property with a value of 44.50. Notice the `<beans>` element at the root of the XML file. This is the root element of any Spring configuration file. The `<bean>` element is used to tell the Spring container about a class and how it should be configured. The `id` attribute is used to name the bean `currencyConverter` and the `class` attribute specifies the bean's fully qualified class name.

Within the `<bean>` element, the `<property>` element is used to set a property, in this case `exchangeRate` property. By using `<property>`, we are telling the Spring container to call `setExchangeRate()` when setting the property. This is called **setter injection** and is a straightforward way to configure and wire bean properties. The value of the exchange rate is defined using the `value` attribute. The following snippet of code illustrates roughly what the container does when instantiating the `currencyConverter` service based on the XML definition seen above.

```
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl();
currencyConverter.setExchangeRate(44.50);
```

Notice the configuration metadata is represented in XML. But it can also be done using Java annotations, or Java code.



1.3 : Spring Jumpstart

Injecting dependencies via setter methods

The diagram illustrates the execution flow. On the left, the Java code for `CurrencyConverterClient` is shown. A callout bubble labeled "The client application" points to the code. An arrow points from the code to a callout bubble on the right labeled "Output". The output bubble contains the command-line output of the application.

`public class CurrencyConverterClient {
 public static void main(String args[]) throws Exception {
 Resource res = new ClassPathResource("currencyconverter.xml");
 BeanFactory factory = new XmlBeanFactory(res);
 CurrencyConverter curr = (CurrencyConverter)
 factory.getBean("currencyConverter");
 double rupees = curr.dollarsToRupees(50.0);
 System.out.println("50 $ is "+rupees+" Rs.");
 }
}`

Output:
CurrencyConverterImpl()
setExchangeRate()
dollarsToRupees()
50 \$ is 2225.0 Rs.

iGATE
ITOPS for Business Outcomes

LEARN
iGATE Learning, Education and Research Network

Finally look at the class above. This class loads the Spring container and uses it to retrieve the currencyConverter service. The BeanFactory class used here is the Spring container. After loading the currencyconverter.xml file into the container, the main() method calls the getBean() method on the BeanFactory to retrieve a reference to the CurrencyConverter service. With this reference in hand, it finally calls the dollarsToRupees() method. When we run the above application (`CurrencyConverterClient.java`), output is as seen above.

This example illustrates the basics of configuring and using a class in Spring. It is simple because it only illustrates how to configure a bean by injecting a double value into a property. The real power of Spring lies in how beans can be injected into other beans using IoC (Inversion of Control – which shall be discussed in the next topic).

1.3 : Spring Jumpstart
DemoSpring_1



- This demo illustrates how the container will instantiate the CurrencyConverter service using setter injection.



- 14 -

Please refer to demo, DemoSpring_1.

1.3 : Spring Jumpstart

Injecting dependencies via constructor



- Bean classes can be programmed with constructors that take enough arguments to fully define the bean at instantiation

```
<bean id="currencyConverter"
      class="training.Spring.CurrencyConverterImpl">
    <constructor-arg>
      <value> 44.50 </value>
    </constructor-arg>
</bean>
```

```
public CurrencyConverterImpl(double er) {
    exchangeRate = er;
}
```

- 15 -



Injecting dependencies via constructor:

Setter injection assumes that all mutable properties are available via a setter method. For one thing, when this type of bean is instantiated, none of its properties have been set and it could possibly be in a invalid state. Second, you may want all properties to be set just once, when the bean is created and become immutable after that point. This is impossible when all properties are exposed via setter methods.

In Java, a class can have multiple constructors and thus you can program your bean classes with constructors that take enough arguments to fully define the bean at instantiation. This is called constructor injection. In the above example, this is done by having Spring set the exchangeRate property through currencyConverterImpl's single argument constructor.

1.3 : Spring Jumpstart

Injecting dependencies via constructor



- If a constructor has multiple arguments, then ambiguities among constructor arguments can be dealt with in two ways :
 - by index
 - by type

```
<beans>
<bean id="currencyConverter"
      class="training.Spring.CurrencyConverterImpl3">
<constructor-arg><value>44.25</value></constructor-arg>
<!-<constructor-arg index="0"><value>44.25</value></constructor-arg>>
<!-<constructor-arg type="double"><value>44.25</value></constructor-arg>>
</bean>
</beans>
```



- 16 -

The `<constructor-arg>` element has an optional `index` attribute that specifies the ordering of the constructor arguments.

```
<constructor-arg index="1">
  <value> some-value </value>
<constructor-arg>
```

The `type` attribute lets you specify exactly what type each argument is supposed to be.

```
<constructor-arg type="java.lang.String">
  <value> some-value </value>
<constructor-arg>
```

The code above illustrates how the container will instantiate the `CurrencyConverter` service when using the `<constructor-arg>` element using the same classes seen earlier.

1.3 : Spring Jumpstart
DemoSpring_2



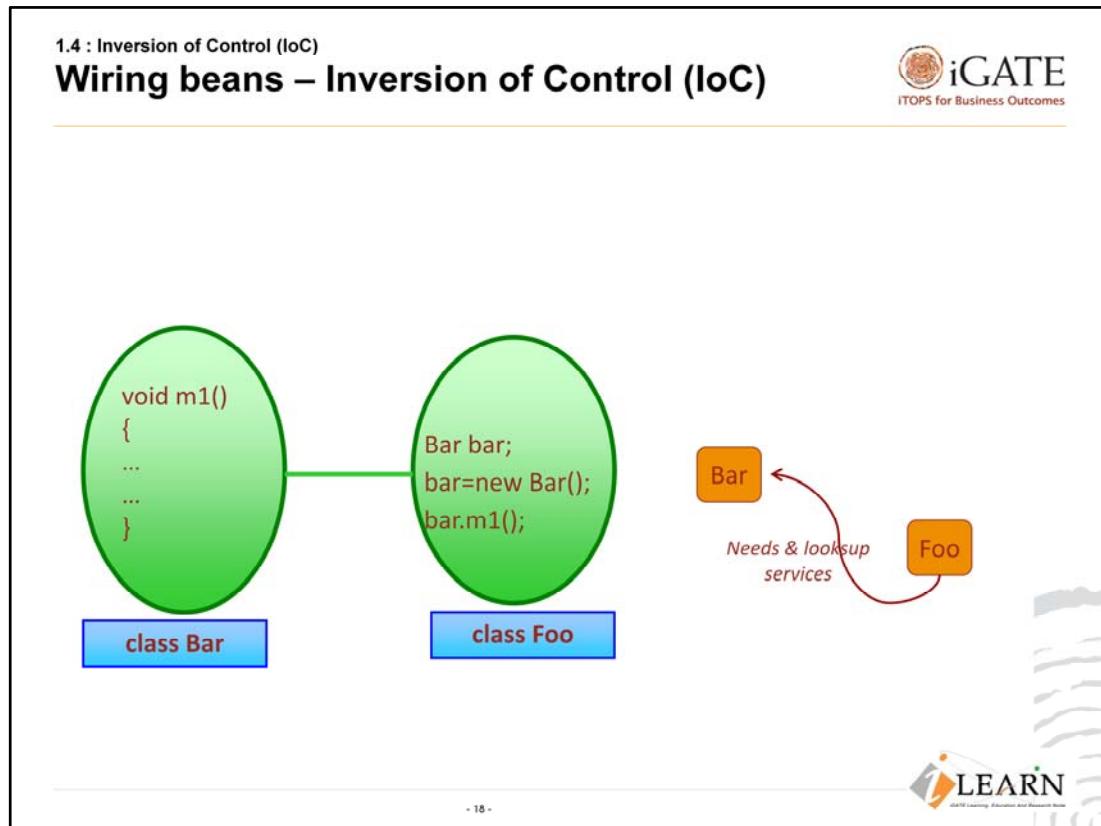
- This demo illustrates how the container will instantiate the CurrencyConverter service when using the <constructor-arg> element.



- 17 -

Please refer to demo, **DemoSpring_2**.

Sometimes the only way to instantiate an object is through a static factory method. Spring is ready-made to wire factory-created beans through the <bean> element's factory-method attribute. DemoSpring_2 demonstrates this. Pls. refer to demos.



Software code is normally broken down into logical components or services that interact with one another. In Java, these components are usually instances of Java classes or objects. Each object must use or work with other objects in order to do its job.

To understand this better, let us see a situation in which two Java classes need to communicate. In the above figure, class Foo depends on an instance of class Bar for some functionality. Traditionally, Foo creates instance of Bar using new operator or obtains one from a factory class.

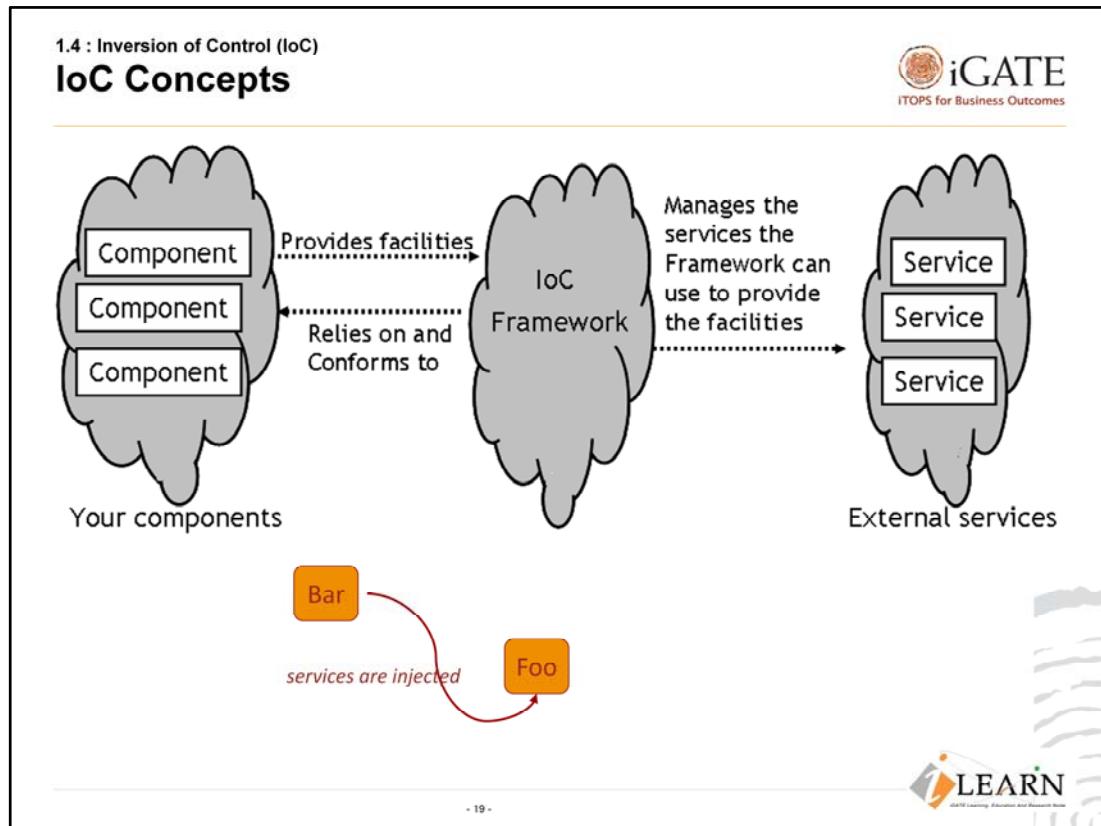
In IoC however, an instance of Bar is provided to Foo at runtime by some external processes i.e. the container will handle the “injection” of an appropriate implementation.

Inversion of Control is best understood through the term the “Hollywood Principle,” which basically means “Don’t call me, I’ll call you.”

We don’t directly connect our components and services together in code but describe which services are needed by which components in a configuration file. A container is responsible for hooking it up. This concept is similar to ‘Declarative Management’.

Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application.

Difference lies in how these classes are configured and introduced to each other. Typically a Spring application has an XML (the Spring configuration) file that describes how to configure these classes.



Understanding inversion of control: Inversion of control is at the heart of the Spring framework. As seen earlier, any non-trivial application is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to a highly coupled and hard-to-test code.

Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system ie dependencies are *injected* into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

Dependency injection is kind of an Inversion of Control pattern. The term dependency injection describes the process of providing (or injecting) a component with the dependencies it needs, in an IoC fashion. Dependency Injection proposes separating the implementation of an object and the construction of objects that depend on them.

Dependency injection is a form of PUSH configuration; the container pushes dependencies into application objects at runtime. This is opposite to the traditional PULL configuration in which the app object pulls dependencies from its environment.

In the figure, we have application objects offering external services. The application components depend on these external services. The job of coordinating the implementation and construction is left to the assembler code, which in this case would be the spring IoC framework.

1.4 : Inversion of Control (IoC)

IoC, Beans and BeanFactories



- Used to achieve loose coupling between several interacting components in an application.
- The IoC framework separates facilities that your components are dependent upon and provides the “glue” for connecting the components.
- BeanFactory is the core of Spring’s DI container.
- In Spring, the term “bean” is used to refer to any component managed by the container.

- 20 -



Loose coupling is one of the critical elements in object-oriented software development. It allows you to change the implementations of two related objects without affecting the other object. Strong coupling directly affects scalability of an application. Tightly coupled code is difficult to test, reuse and understand. On the other hand, completely uncoupled code doesn’t do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but it must be managed very carefully. A common technique used to reduce coupling is to hide implementation details behind interfaces so that actual implementation class can be swapped out without impacting the client class. That is what IoC is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves. This is where lightweight framework containers like Spring come into play. IoC introduces the concept of a framework of components that in turn has many similarities to a J2EE container. The IoC framework separates facilities that your components are dependent upon and provides the “glue” for connecting the components.

With Inversion of Control (IoC), you can achieve loose coupling between several interacting components in an application.

The core of Spring’s DI container is the BeanFactory. A bean factory is responsible for managing components and their dependencies. We shall see bean factories in detail later in this session. In Spring, the term “bean” is used to refer to any component managed by the container. Typically, beans adhere to Javabeans specification



1.4 : Inversion of Control (IoC)

Inversion of Control Approaches



- **IoC pattern uses three different approaches to achieve decoupling of control of services from components:**
 - Type 1 : Interface injection
 - Type 2 : Setter Injection
 - Type 3 : Constructor injection

- 21 -



The IoC pattern uses three different approaches to achieve decoupling of control of services from your components:

1. **Type 1 : Interface injection:** This is how most J2EE worked. Components are explicitly conformed to a set of interfaces with associated configuration metadata, in order to allow framework to manage them correctly.
2. **Type 2 : Setter Injection:** External metadata is used to configure how components can interact. Our first example used this approach, by using a Springconfig.xml file.
3. **Type 3 : Constructor injection:** Components are registered with the framework, including the parameters to be used when the components are constructed, and the framework provides instances of the component with all the specified facilities applied. Our last example used this approach.

There is a fourth approach called "**Lookup-method injection**". This has been covered in detail in Appendix-B.

1.4 : Inversion of Control (IoC)

IoC in action: Wiring Beans



- The act of creating associations between application components is known as **wiring**.
- In Spring, there are many ways of wiring components together, but most commonly used is XML. An example:

```
<bean id="exchangeService" class="ExchangeServiceImpl" />
<bean id="currencyConverter" class="CurrencyConverterImpl">
    <property name="exchangeService">
        <ref bean="exchangeService" />
    </property>
    <!--<property name="exchangeService">
        <ref local="exchangeService" /></property> -->
    <!--<property name="exchangeService">
        <idref local="exchangeService" /></property> -->
</bean>
```

- 22 -



The act of creating associations between application components is known as **wiring**. In Spring there are many ways of wiring components together, but most commonly used is XML. A BeanFactory will load the bean definition and wire the beans together.

In the above example, the `<ref>` subelement of the `<property>` tag is used to set the value or constructor argument to be a reference to another bean from the factory. The `bean` attribute is the ID of the other bean. Specifying the target bean by using the `bean` attribute of the `ref` tag is the most general form, and will allow creating a reference to any bean in the same BeanFactory or parent BeanFactory.

Specifying the target bean by using the `local` attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the `local` attribute must be the same as the `id` attribute of the target bean. The XML parser will issue an error if no matching element is found in the same file. As such, using the `local` variant is the best choice (in order to know about errors as early as possible) if the target bean is in the same XML file.

Specifying the `idref` tag will allow Spring to validate at deployment time whether the other bean actually exists.

1.4 : Inversion of Control (IoC)

DemoSpring_3



- This demo illustrates how the BeanFactory loads the bean definition and wires the beans together



- 23 -

Please refer to demo, **DemoSpring_3**. CurrencyConverterClient.java uses an XmlBeanFactory (a BeanFactory implementation) to load currencyconverter.xml and to get a reference to the CurrencyConverter object. It then invokes the dollarsToRupees() method.

There are two application objects. The instance of ExchangeRateImpl is responsible for retrieving the exchange rate, using which, the currency converter instance would convert currency. When the CurrencyConverterImpl is instantiated, it in turn first instantiates the ExchangeService bean. The ExchangeService instance in the CurrencyConverterImpl class then exposes its getExchangeRate() method to return the exchange rate.

Summarizing how the container initializes and resolves bean dependencies:

- The container first initializes the bean definition, without initializing the bean itself, typically at the time of the container startup. The bean dependencies may be explicitly expressed in the form of constructor arguments or arguments to a factory method and/or bean properties.
- Each property or constructor argument in a bean definition is either an actual value to set, or a reference to another bean in the bean factory.
- Constructor arguments or bean properties that refer to another bean will force the container to create or obtain that other bean first. Effectively, the referred bean is a dependent of the calling bean. This can trigger a chain of bean creation.
- Every Constructor argument or bean property must be able to be converted from String format to the actual format expected. Spring is able to convert from String to built-in scalar types like int, float etc. *Spring uses JavaBeans PropertyEditors to convert all other types.*

1.4 : Inversion of Control (IoC)

Prototyping Vs Singleton



- By default, all Spring beans are singletons.
- But each time a bean is asked for, prototyping lets the container return a new instance.
- This is achieved through the scope attribute of <bean>
- Example:

```
<bean id="foo" class="com.patni.Foo" scope="prototype" />
```

- Additional Bean scopes:

- request
- session
- global-session

- 24 -



Singleton beans, the default, are created only once by the container and all calls to BeanFactory.getBean() return the same instance. The container will then hold and use the same instance of the bean whenever it is referenced again. This can be significantly less expensive in terms of resource usage than creating a new instance of the bean on each request.

A non-singleton, or prototype bean, may be specified by setting the scope attribute to `prototype` (see example above). The lifecycle of a prototype bean will often be different than a singleton. When a container is asked to supply a prototype bean, it's initialized and then used, but the container does not hold on to it past that point.

Prototyped beans are useful when you want the container to give a unique instance of a bean each time it is asked for, but you still want to configure one or more properties of the bean through Spring. Thus a new instance is created when `getBean()` is invoked with the bean's name.

Previous versions of Spring had IoC container level support for exactly two distinct bean scopes (singleton and prototype). Spring 2.0 onwards provides a number of additional scopes depending on the environment in which Spring is being deployed (for example, request and session scoped beans in a web environment).

It also provides integration points so that Spring users can create their own scopes. Beans can be defined to be deployed in one of a number of scopes. See the table in the next page for the different scopes.

Scope	What it does
singleton	Scopes the bean definition to a single instance per Spring container (default).
prototype	Allows a bean to be instantiated any number of times (once per use).
request	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
session	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
global-session	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

An example:

```
<bean id="accountService" class="com.foo.DefaultAccountService"  
      scope="singleton"/>
```

1.4 : Inversion of Control (IoC)

Inner beans

- Another way of wiring bean references is to embed a `<bean>` element directly in the `<property>` element

```
<bean id="currencyConverter" class="CurrencyConverterImpl">
    <property name="exchangeService">
        <bean class= "ExchangeServiceImpl" />
    </property>
</bean>
```

- The drawback here is that the instance of inner class cannot be used anywhere else; it is an instance created specifically for use by the outer bean.



- 26 -

The drawback of the above method is that you cannot reuse the instance of `ExchangeServiceImpl` anywhere else – it is an instance created specifically for use by the `currencyConverter` bean.

Inner beans aren't limited to setter injection. You may also wire inner beans into constructor arguments. E.g.:

```
<bean id="currencyConverter"
      class="com.Spring.CurrencyConverterImpl4">
    <constructor-arg>
        <bean class="com.Spring.ExchangeServiceImpl" />
    </constructor-arg>
</bean>
```

Autowiring

- Autowiring allows Spring to wire all bean's properties automatically by setting the autowire property on each <bean> that you want autowired

```
<bean id="foo" class="com.Patni.Foo" autowire="autowire type" />
```

- Four types of autowiring:

- byName
- byType
- constructor
- Autodetect

So far, you have seen how to wire all your bean's properties explicitly. You can also have Spring wire them automatically by setting the autowire property on each bean that you want autowired.

There are four types of autowiring:

1. **byName**: Attempts to find a bean in the container whose name (or id) is same as the name of the property being wired. If matching bean is not found, then property will remain unwired.
2. **byType**: Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties. Properties for which there's no matching bean will remain unwired.
3. **constructor** : Tries to match a constructor of the autowired bean with beans whose types are assignable to the constructor arguments. In the event of ambiguous beans or ambiguous constructors, an org.springframework.beans.factory.UnsatisfiedDependencyException will be thrown.
4. **autodetect** : Attempts constructor autowiring first. If that fails, attempts autowiring byType.

1.4 : Inversion of Control (IoC)

DemoSpring_4



- This demo illustrates automatically wiring your beans



- 28 -

Refer to **demo, DemoSpring_4**. This uses the same exchange service and currency converter service seen in earlier examples. However, by using the **autowire** attribute in the currencyconverter.xml we can execute the client program even without specifying the exchange service.

```
<bean id="exchangeService" class="com.igate.intro.ExchangeService">
    <constructor-arg><value>44.25</value></constructor-arg>
</bean>
```

```
<bean id="currencyConverter"
      class="com.igate.intro.CurrencyConverter" autowire="byName"/>
```

```
public class CurrencyConverter
{
    private ExchangeService exchangeService;
    .....
}
```

1.4 : Inversion of Control (IoC)

Using collections for injection



- Often, beans need access to collections of objects, rather than just individual beans or values.
- Spring allows you to inject a collection of objects into your beans.
- You can choose either <list>, <map>, <set> or <props> to represent a List, Map, Set or Properties instance.
- You will pass in the individual items just as you would with any other injection.



- 29 -

Example:

```
<bean id="complexObject" class="example.ComplexObject">
    <property name="people">
        <props>
            <prop key="HarryPotter">The magic property</prop>
            <prop key="JerrySeinfeld">The funny property</prop>
        </props>
    </property>
    <property name="someList">
        <list>
            <value>red</value>
            <value>blue</value>
        </list>
    </property>
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key="a ref" value-ref="myDataSource"/>
        </map>
    </property>
</bean>
```

One example is also available. Refer to [demo](#), [DemoSpring_5](#)

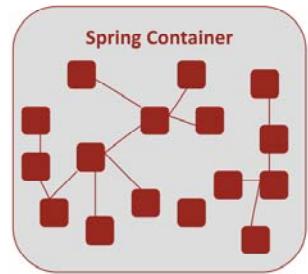


1.5 : Bean containers

Bean containers: concept



- The container or bean factory is at the core of the Spring framework and uses IoC to manage components.
- Bean factory is responsible to create and dispense beans.
- It takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.
- Spring has two types of containers:
 - Bean factories that are the simplest, providing basic support for dependency injection
 - Application contexts that build on bean factory by providing application framework services



- 30 -

The container is at the core of the Spring framework and uses IoC to manage components. The basic IoC container in Spring is called the bean factory. Any bean factory allows the configuration and wiring of objects using dependency injection, in a consistent and workable fashion. A bean factory also provides some management of these objects, with respect to their lifecycles. Thus,

- Bean factory is a class whose responsibility is to create and dispense beans.
- A bean factory knows about many objects within an application.
- Able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from the bean itself and the bean's client. As a result, when a bean factory hands out objects, those objects are fully configured, aware of their collaborating objects and ready to use.
- A bean factory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.

Spring actually comes with two different types of containers:

Beanfactory interface: provides an advanced configuration mechanism capable of managing any type of object.

ApplicationContext interface : is a sub-interface of BeanFactory. It allows easier integration with Spring's AOP features, message resource handling, event publication, and application-layer specific contexts such as the WebApplicationContext for use in web applications.

We shall look at the Application context in detail later.

1.5 : Bean containers

Bean containers: The BeanFactory



- **BeanFactory interface is responsible for managing beans and their dependencies**
- **Its getBean() method allows you to get a bean from the container by name**
- **It has a number of implementing classes:**
 - DefaultListableBeanFactory
 - SimpleJndiBeanFactory
 - StaticListableBeanFactory
 - XmlBeanFactory

- 31 -



Lets start our exploration of Spring containers with the most basic of Spring containers : the BeanFactory.

Bean factory is responsible for managing beans and their dependencies. Your application interacts with Spring DI container via the BeanFactory interface. It has a getBean() method that allows you to get a bean from the container by name. Additional methods allow you to query the bean factory to see if bean exists, to find the type of bean and to find if a bean is configured as a singleton.

Different bean factory implementations exist to support varying levels of functionality with XmlBeanFactory being the most common representation.

A partial listing of the impelmenting classes follows:

- DefaultListableBeanFactory
- SimpleJndiBeanFactory
- StaticListableBeanFactory
- XmlBeanFactory



Please refer to the Spring documentation for more information on these classes

1.5 : Bean containers

The XmlBeanFactory



- One of the most useful implementations of the bean factory is instantiated via explicit user code as:

```
Resource res = new FileSystemResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
Resource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

- 32 -



One of the most useful implementations of the bean factory is the `org.springframework.beans.factory.xml.XmlBeanFactory`. The BeanFactory is instantiated via explicit user code such as shown above.

This simple line of code tells the bean factory to read the bean definitions from the XML file (`beans.xml` in this case). But the bean factory doesn't instantiate the beans just yet. Beans are "lazily" instantiated into bean factories, meaning that while the bean factory will immediately load the bean definitions, beans themselves will not be instantiated until they are needed.

The Spring IoC container consumes some form of *configuration metadata*; which is nothing more than how you 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 its stuff.

Note

XML-based metadata is by far the most commonly used form of configuration metadata. It is *not* however the only form of configuration metadata that is allowed. The Spring IoC container itself is *totally* decoupled from the format in which this configuration metadata is actually written. The XML-based configuration metadata format really is simple though, and so the majority of this material will use the XML format to convey key concepts and features of the Spring IoC container.

1.5 : Bean containers

The Resource interface



➤ **The Resource interface is a unified mechanism for accessing resources in a protocol-independent manner.**

➤ **Some methods:**

- `getInputStream()`: locates and opens the resource, returning an `InputStream` for reading from the resource
- `exists()`: indicates whether this resource actually exists
- `isOpen()`: indicates whether this resource represents a handle with an open stream
- `getDescription()`: returns a description for this resource, to be used for error output

- 33 -



The Resource interface: Often, an application needs to access a variety of resources in different forms. You may need to access some configuration data stored in a file in the filesystem, some image data stored in a JAR file on the classpath, or maybe some data on a server elsewhere. Spring provides a unified mechanism for accessing resources in a protocol-independent manner. This means that your application can access a file resource in the same way, whether it is stored in the file system, the classpath or on a remote server.

At the core of the Spring's support is the Resource interface. This defines self-explanatory methods mentioned above. There are a number of implementations that come supplied straight out of the box in Spring:

UrlResource : The UrlResource wraps a `java.net.URL`, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc.

ClassPathResource : This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.

FileSystemResource : This is a Resource implementation for `java.io.File` handles. It obviously supports resolution as a `File`, and as a URL.

ServletContextResource: This is a Resource implementation for `ServletContext` resources, interpreting relative paths within the relevant web application's root directory.

Two of the implemented classes examples for this interface ie. `ClassPathResource` and `FileSystemResource`, are shown in previous slide. Please refer to the Spring documentation for more details.

1.5 : Bean containers

The XmlBeanFactory (Cont...)



- In an XmlBeanFactory, bean definitions are configured as one or more bean elements inside a top-level beans element

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="..." class="...">
        ...
    </bean>
    ...
</beans>
```

- 34 -

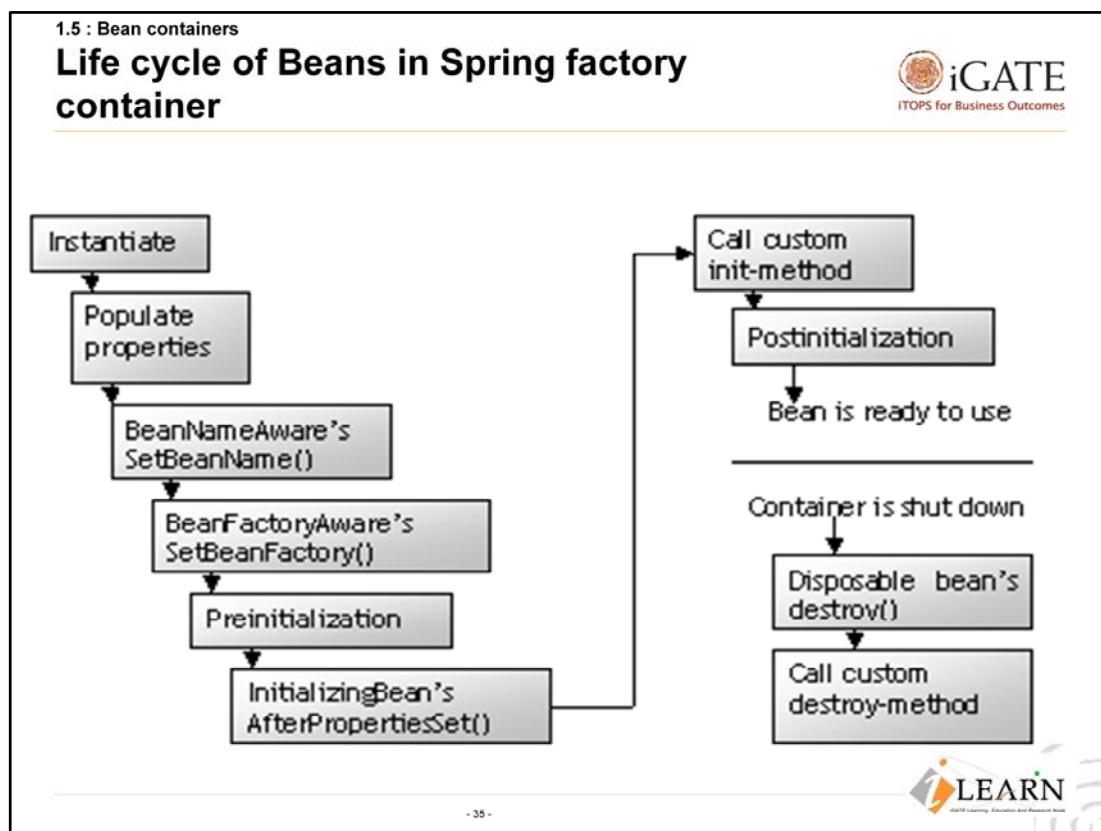


A BeanFactory configuration consists of, at its most basic level, definitions of one or more beans that the BeanFactory must manage. In an XmlBeanFactory, these are configured as one or more bean elements inside a top-level beans element. The first versions of Spring used a DTD. But Spring 2.0 onwards uses schema for the xml configuration file.

To retrieve a bean from a bean factory, simply call the getBean() method, passing it the name of the bean you want to retrieve.

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

When getBean() is called, the factory will instantiate the bean and begin setting the bean's properties using dependency injection. Thus begins the bean's life cycle within the container (explained further on).



In a traditional Java application, the life cycle of a bean is fairly simple. Java's `new` keyword is used to instantiate the bean and it is ready to use. In contrast, the life cycle of a bean within a Spring container is a bit more elaborate.

A bean factory performs several setup steps before a bean is ready to use.

- The container finds the bean's definition and instantiates the bean.
- Using dependency injection, Spring populates all the properties as specified in the bean definition.
- If the bean implements the `BeanNameAware` interface, the factory calls `setBeanName()` passing the bean's ID.
- If the bean implements the `BeanFactoryAware` interface, the factory calls `setBeanFactory()` passing an instance of itself.
- If there are any `BeanPostProcessors` associated with the bean, their `PostProcessBeforeInitialization()` methods will be called.
- If an init-method is specified for the bean, it will be called.
- Finally, if there are any `BeanPostProcessors` associated with the bean, their `PostProcessAfterInitialization()` methods will be called.
- The bean is now ready to be used and will remain in the bean factory until it is no longer needed. It is removed from the factory in two ways:
 - ✓ If the bean implements the `DisposableBean` interface, the `destroy()` method is called.
 - ✓ If a custom destroy-method is specified, it will be called.

1.5 : Bean containers

Initialization and Destruction



- When a bean is instantiated, some initialization can be performed to get it to a usable state
- When the bean is removed from the container, some cleanup may be required
- Spring can use two life-cycle methods of each bean to perform this setup and teardown.
- Example:

```
<bean id="foo" class="com.spring.Foo"  
      init-method="setup"  
      destroy-method="teardown" />
```



- 36 -

Declaring a custom init-method in your bean's definition specifies a method that is to be called on the bean immediately upon instantiation. Similarly, a custom destroy-method specifies a method that is called just before a bean is removed from the container.

E.g., the init-method="setup" in the above example calls setup() method in the bean class when bean is loaded into container and teardown() when bean is removed from container.

Defaulting init-method and destroy-method:

If many of the beans in a context definition file will have initialization or destroy methods with same name, you don't have to declare init-method or destroy-method on each individual bean. Instead, you can take advantage of the default-init-method and default-destroy-method attributes on the <beans> element.

```
<beans.....  
  default-init-method="tuneApplication"  
  default-destroy-method="cleanApplication" >  
  .....  
</beans>
```

1.5 : Bean containers

InitializingBean and DisposableBean



➤ InitializingBean interface

- provides afterPropertiesSet() method which is called once all specified properties for the bean have been set.

➤ DisposableBean interface

- provides destroy() method which is called when the bean is disposed by the container

➤ **The advantage is that Spring container is able to automatically detect beans without any external configuration.**

➤ **The drawback is that the applications' beans are coupled to Spring API.**

- 37 -



As an option to init-method and destroy-method, we can also rewrite the bean class to implement two special Spring interfaces: InitializingBean and DisposableBean. The Spring container treats beans that implement these interfaces in a special way, by allowing them to hook into the bean lifecycle. The InitializingBean interface provides a method afterPropertiesSet(). This is called once all specified properties for the bean have been set. This makes it possible for the bean to perform initialization that cannot be performed until all properties have been completely set.

DisposableBean provides a destroy() method which will be called on the other end of a bean-lifecycle., when the bean is disposed by the container.

The benefit of using these interfaces is that Spring container is able to automatically detect beans that implement them without any external configuration. However, the disadvantage of implementing these interfaces is that you couple your application's beans to Spring API.

```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.DisposableBean;
public class SampleBean implements InitializingBean, DisposableBean{
    ...
    public void afterPropertiesSet(){...}
    public void destroy(){...}
}
```

1.5 : Bean containers

Bean containers:Application context



➤ **Provides application framework services such as :**

- Resolving text messages, including support for internationalization of these messages
- Load file resources, such as images
- Publish events to beans that are registered as listeners

➤ **Many implementations of application context exist:**

- ClassPathXmlApplicationContext
- FileSystemApplicationContext
- XmlWebApplicationContext

- 38 -



Application contexts build on bean factory by providing application framework services. A bean factory is fine for simple applications, but to take advantage of the full power of Spring framework, we need to use the application context container, which offers services mentioned in slide above. Many implementations of application context exist:

- **ClassPathXmlApplicationContext** : Loads context definition from a XML file located in the class path.
- **FileSystemApplicationContext**: Loads context definition from an XML file in the file system.
- **XmlWebApplicationContext** : Loads context definition from an XML file contained within a web application

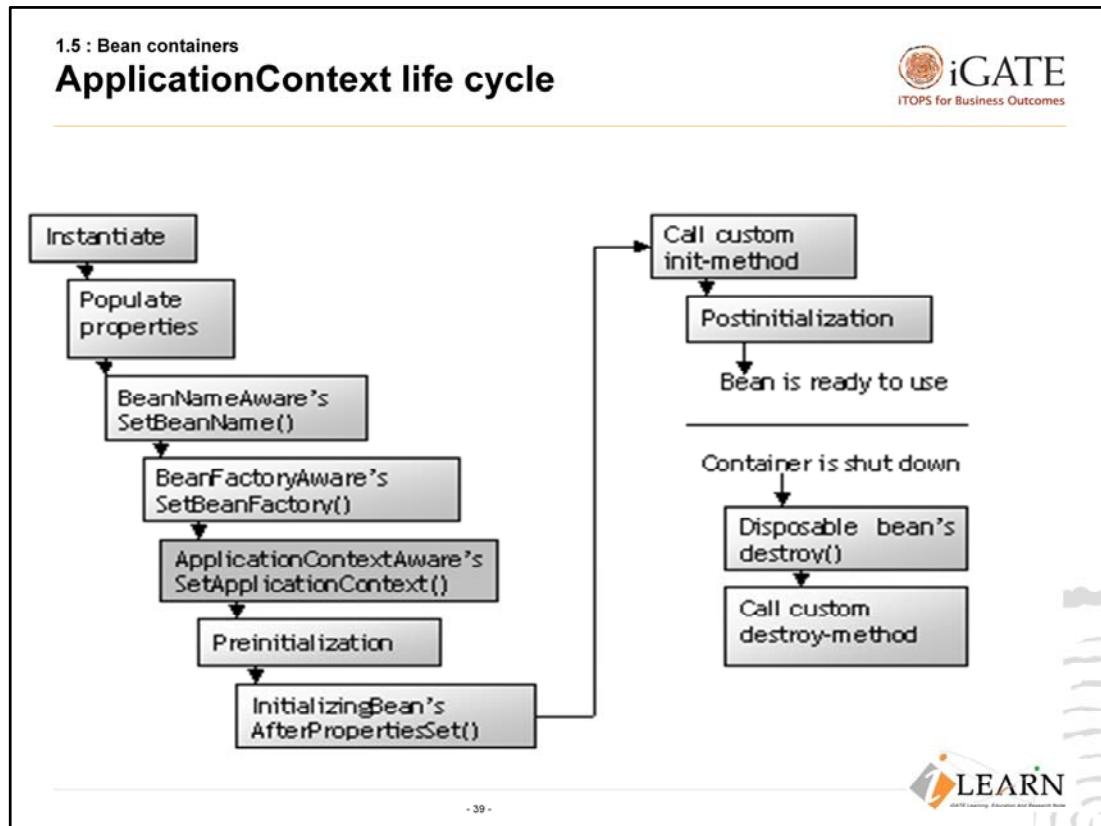
Loading the ApplicationContext: Some examples :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("app.xml");
ApplicationContext ctx = new
```

```
    FileSystemXmlApplicationContext("/some/file/path/app.xml");
    ApplicationContext ctx = new ClassPathXmlApplicationContext( new
        String[]{"app1.xml","app2.xml"}); // combines multiple xml file fragments
```

Spring support Wildcards in application context constructor resource paths.

The resource paths in application context constructor values may be a simple path which has a one-to-one mapping to a target Resource, or alternately may contain the special "classpath*:" prefix and/or internal Ant-style regular expressions (matched using Spring's PathMatcher utility). Both of the latter are effectively wildcards. (Continued on next page....)



(Continued from previous page)

Eg.

```

/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
classpath:com/mycompany/**/service-context.xml
ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath*:conf/appContext*.xml");
  
```

ApplicationContext life cycle:

The life cycle of a bean within a Spring ApplicationContext differs only slightly from that of a bean within a bean factory as shown in above figure.

The only difference is that if a bean implements the ApplicationContextAware interface, the setApplicationContext() method is invoked.

1.5 : Bean containers

Customizing beans with BeanPostProcessor



- Post processing involves cutting into a bean's life cycle and reviewing or altering its configuration.
- Occurs after some event has occurred.
- Spring provides two interfaces :
 - BeanPostProcessor interface
 - BeanFactoryPostProcessor interface
- ApplicationContext automatically detects Bean Post-Processor, but these have to manually be explicitly registered for bean factory.

- 40 -



The lifecycle of the BeanFactory and ApplicationContext provide many opportunities to cut into the bean's life cycle to review or alter its configuration. This is called **post processing** and occurs *after* some event has occurred. A bean post-processor is a java class which implements the BeanPostProcessor interface, which consists of two callback methods:

- postProcessBeforeInitialization: called immediately before bean initialization.
- postProcessAfterInitialization: called immediately after bean Initialization.

BeanPostProcessors operate on bean (or object) *instances*; ie the Spring IoC container instantiates a bean instance and *then* BeanPostProcessor interfaces do their work.

An ApplicationContext will automatically detect any beans which are deployed into it which implement the BeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the factory on bean creation. Simply deploy the post-processor in a similar fashion to any other bean! However, for BeanFactory, bean post-processors have to manually be explicitly registered, with a code sequence as shown below.

Since this manual registration step is not convenient, and ApplicationContexts are functionally supersets of BeanFactories, it is generally recommended that ApplicationContext variants are used when bean post-processors are needed.

```
ConfigurableBeanFactory bf = new .....; // create BeanFactory  
// now register some beans and any needed BeanPostProcessors  
MyBeanPostProcessor pp = new MyBeanPostProcessor();  
bf.addBeanPostProcessor(pp); // now start using the factory ...
```

1.5 : Bean containers

Customizing beans with BeanFactoryPostProcessor



- BeanFactoryPostProcessor performs post processing on the entire Spring container.
- It has a single method, which is postProcessBeanFactory().
- Spring offers a number of pre-existing bean factory post-processors:
 - AspectJWeaving
 - CustomAutowireConfigurer
 - CustomEditorConfigurer
 - CustomScopeConfigurer
 - PropertyPlaceholderConfigurer
 - PreferencesPlaceholderConfigurer
 - PropertyOverrideConfigurer



- 41 -

BeanFactoryPostProcessors operate on the *bean configuration metadata*; that is, the Spring IoC container allows BeanFactoryPostProcessors to read the configuration metadata and potentially change it *before* the container instantiates any beans other than BeanFactoryPostProcessors!

Has a single method – postProcessBeanFactory(). This is called by Spring container after all bean definitions have been loaded but before any beans are instantiated (including BeanPostProcessor beans).

Spring offers a number of pre-existing bean factory post-processors. Two very useful implementations are:

- **PropertyPlaceholderConfigurer** : Loads properties from one or more external property files and uses these properties to fill in place holder variables in the bean wiring XML file.
- **CustomEditorConfigurer** : Lets you register custom implementation of java.beans.PropertyEditor to translate property wired values to other property types.

Let's take a look at how you can use these implementations of BeanFactoryPostProcessor.

1.5 : Bean containers

PropertyPlaceholderConfigurer



- It is possible to configure entire application in a single bean wiring file.

```
bean id="datasource" class="com.spring.ConnectionDataSource" >
    <property name="url">
        <value> jdbc:hsqldb:training </value>
    </property>
    <property name="driverclassname">
        <value> org.hsqldb.jdbcDriver </value>
    </property>
    ...
</bean>
```

- But, sometimes it is beneficial to extract certain pieces of that configuration into a separate property file.



For the most part, it is possible to configure entire application in a single bean wiring file. But sometimes it is beneficial to extract certain pieces of that configuration into a separate property file. E.g. , a configuration concern common to many applications is configuring a data source. Traditionally, in Spring, you do this with the following XML in the bean wiring file (see above) Configuring the data source directly in the bean wiring file may not be appropriate. The database specifics are a deployment detail and must be separated.

1.5 : Bean containers

PropertyPlaceholderConfigurer



- Externalizing properties using **PropertyPlaceholderConfigurer** indicates Spring to load certain configuration from an external property file.

```
<bean id="placeHolderConfig" class="org.springframework.beans.  
factory.config.PropertyPlaceholderConfigurer">  
    <property name="location" value="data.properties" />  
</bean>  
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}" />  
    <property name="url" value="${jdbc.url}" />  
    <property name="username" value="${jdbc.username}" />  
    <property name="password" value="${jdbc.password}" />  
</bean>
```

jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@192.168.224.26:1521:trgdb
.....

- 43 -



Fortunately, externalizing properties in Spring is easy if you are using ApplicationContext as your Spring container. You can use PropertyPlaceholderConfigurer to tell Spring to load certain configuration from an external property file as shown in the code snippet above. The **location** property tells Spring where to find the property file data.properties. When Spring creates the bean, the PropertyPlaceholderConfigurer will step in and replace the place holder variables with the values from the property file. It *pulls* values from a properties file into bean definitions.

1.5 : Bean containers

Demo: DemoSpring_6



- This demo shows how to use the **PropertyPlaceholderConfigurer** **BeanFactoryPostProcessor**



< 44 >

Please refer to **demo, DemoSpring_6**. In this case user.java is a POJO with two properties – username and password. We shall set the properties of this bean during its instantiation using external properties file. user.properties is a properties file. user.xml has two place holder variables \${username} and \${password}

```
<bean id="user" class="training.spring.User">
    <property name="username"><value>${username}</value></property>
    <property name="password"><value>${password}</value></property>
</bean>
```

Whenever setter is called, the listener (PropertyPlaceholderConfigurer) is invoked and it will look into the properties file, retrieve values, place them in the place holders and initialize.

If instead of application context, bean factory is used, then the listener would have to be explicitly registered as shown below (also in the commented out code in userClient.java file).

```
XmlBeanFactory factory = new XmlBeanFactory(new
    FileSystemResource("user.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("user.properties"));
cfg.postProcessBeanFactory(factory);
```

1.5 : Bean containers

CustomEditorConfigurer



- **CustomEditorConfigurer** is a bean factory post-processor which allows to convert values in String form to final property values.
- It allows you to register custom implementation of **PropertyEditor** to translate property wired values to other property types.
- **Java.beans.PropertyEditorSupport** is a convenience implementation **java.beans.PropertyEditor** interface that allows setting a non-string property to a string value.
- It has two methods: **getAsText()** and **setAsText(String s)**

- 45 -



CustomEditorConfigurer : Lets you register custom implementation of **java.beans.PropertyEditor** to translate property wired values to other property types. This is a bean factory post-processor which allows to convert values in String form to final property values.

So far, we have seen several examples in which a complex property is set with a simple String value. When setting bean properties as a string value, a BeanFactory ultimately uses standard **java.beans.PropertyEditor** interface to convert these strings to the complex type of the property. There is a convenience implementation of the above interface called **java.beans.PropertyEditorSupport**, that has two methods of interest to us:

- **getAsText()** : returns the String representation of a property's value.
- **setAsText(String value)** : sets a bean property value from the string value passed in.

If an attempt is made to set a non-string property to a string value, the **setAsText()** method is called to perform the conversion. Likewise, the **getAsText()** is called to return a textual representation of the property's value. Spring comes with several custom editors based on **PropertyEditorSupport**. You can also write your own custom editor by extending the **PropertyEditorSupport** class.

1.5 : Bean containers

CustomEditorConfigurer



```
<bean id="customEditorConfigurer" class="org.springframework.beans.factory.config.CustomEditorConfigurer">
<property name="customEditors">
    <map>
        <entry key="java.util.Date" value="MyCustomDateEditor"/>
    </map>
</property>
</bean>
```

```
CustomEditorConfigurer configurer = (CustomEditorConfigurer)
    factory.getBean("customEditorConfigurer");
Configurer.postProcessBeanFactory(factory);
BeanClass bean = (BeanClass) factory.getBean("exampleBean");
```



- 46 -

Declarative registration process:

The custom PropertyEditors (MyCustomDateEditor in this case) is injected into the CustomEditorConfigurer class using the Map-typed customEditors property. A map can have multiple entries and each entry in the Map represents a single PropertyEditor with the key of the entry being the name of the class for which the PropertyEditors is used.

In the above first example, the key for the MyCustomDateEditor is java.util.Date, which signifies that this is the class for which the editor should be used.

Programmatic Registration Process:

The second code snippet shows a call to Configurer.postProcessBeanFactory(), which passes in the BeanFactory instance. This is another method of registering custom editors in Spring. You should call this before you attempt to access any beans that need to use the custom PropertyEditors.

However, adding a new PropertyEditor means changing the application code, whereas with the declarative mechanism, you can define your editors as beans, which means you can configure them using DI. Besides, using Application Context means no java code to use the declarative mechanism, which strengthens the argument against programmatic mechanism.

1.5 : Bean containers

Demo: DemoSpring_7



- This demo shows how to use the CustomEditorConfigurer BeanFactoryPostProcessor



< 47 >

Please refer to **DemoSpring_7**. The Employee.java is a POJO that holds the date property. Using basic wiring techniques learnt so far, you could set a value into Employee beans' date property. But we have created SQLDateEditor.java extending PropertyEditorSupport class.

Spring must recognize this custom property editor when wiring bean properties using Spring's CustomEditorConfigurer. This BeanFactory PostProcesser *internally* loads custom editors into the BeanFactory by calling the registerCustomEditor() method. By adding the following bit of XML into the bean configuration file, you will tell Spring to register the SQLDateEditor as a custom editor.

```
<bean id="customEditorConfigurer" class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map><entry key="java.sql.Date">
            <bean class="training.spring.SQLDateEditor" /></entry>
        </map>
    </property>
</bean>
```

You will now be able to configure the Employee objects date property using a simple string value:

```
<bean id="employee" class="training.spring.Employee">
    <property name="date"><value>2006-01-01</value></property>
</bean>
```

1.5 : Bean containers

Internationalization: Resolving text messages



- **ApplicationContext interface provides messaging functionality by extending MessageSource interface.**
- **getMessage() is a basic method used to retrieve a message from the MessageSource.**
- **On loading, ApplicationContext automatically searches for a MessageSource bean defined in the context.**
- **ResourceBundleMessageSource is a ready-to-use implementation of MessageSource.**

- 48 -



Many times you may not want to hard-code certain text that will be displayed to the user. This may be because text is subject to change or perhaps your application will be internationalized and you will display text in the user's native language.

Java's support for parameterization and internationalization of messages enables you to define one or more properties files that contain the text that is to be displayed in your application. There should always be a default message file along with optional language-specific message files. For ex: if name of the application's message bundle is "MsgText", you may have the following set of message property files:

- **MsgText.properties:** Default messages when a locale cannot be determined or locale-specific properties file is not available.
- **MsgText_en_US.properties:** text for English speaking users in US.

Spring's ApplicationContext supports parameterized messages by making them available to the container through the MessageSource interface that has a single method – getMessage(). Spring comes with a ready-to-use implementation of MessageSource – the ResourceBundleMessageSource. This simply uses Java's own java.util. ResourceBundle to resolve messages. A ResourceBundleMessageSource works on a set of properties files that are identified by base names. When looking for a message for a particular Locale, the ResourceBundle looks for a file that is named as a combination of base name and the Locale name. For e.g. : applicationResources_fr will match a French locale.

1.5 : Bean containers

Internationalization: Resolving text messages



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

```
MessageSource messageSource = (MessageSource) factory.getBean  
("messageSource");  
Locale locale = new Locale("en","US");  
String msg = messageSource.getMessage("welcome.message",  
null, locale);
```



< 49 >

To use ResourceBundleMessageSource, add the above xml entry (the first code snippet) to the bean wiring file.

It is very important that this bean be named messageSource because the ApplicationContext will look for a bean specifically by that name when setting up its internal message source.

The basename property specifies the base name of the bundle. The bundle will normally look for messages in properties files with names that are variations of base name depending on locale.

You will never need to inject the messageSource bean into your application beans but will instead access messages via ApplicationContext's own getMessage() methods. For e.g. to retrieve the message whose name is "welcome.message", please refer to the second code snippet above.

You will likely be using parameterized messages in the context of a web application, displaying the text on a web page. In that case, you can use Spring's <spring:message> jsp tag to retrieve messages and need not need directly access the ApplicationContext.

```
< spring:message code="welcome.message" />
```

Alternatively, you can use the second argument of the getMessage() to pass in an array of arguments that will be filled in for params within the message

1.5 : Bean containers

DemoSpringI18N



- This demo shows how to provide messaging functionality in the application context.



- 50 -

Please refer to DemoSpringI18N. There are two resource bundles defined :

applicationResources_en_GB.properties

applicationResources_en_US.properties

Message source has been defined in message.xml

MessageClient.java creates a new locale object and uses the getMessage() to access messages from the appropriate resource bundle based on locale. Notice that the message is parameterized :

welcome.message = Welcome {0}, in UK

Hence we need to send value for {0} parameter in this manner:

```
String msg = messageSource.getMessage("welcome.message", new  
Object[]{"Majrul"},locale);
```

This will set the value of the {0} parameter to "Majrul". We can pass in many parameters for a single message, by using this object array in the getMessage() method.

1.6: Spring Annotations

Annotation-based configuration



➤ **Spring has a number of custom annotations:**

- @Required
- @Autowired
- @Resource
- @PostConstruct
- @PreDestroy

➤ **Annotations to configure beans:**

- @Component
- @Controller
- @Repository
- @Service

➤ **Some other transactions:**

- @Transactional
- @AspectJ

- 51 -



So far, we have configured our beans in the XML configuration files. Starting with Spring 2.0, we can use annotations to configure our beans. By annotating your classes, you state their typical usage or stereotype. Spring has a number of custom (Java5+) annotations.

- **@Required :** The @Required annotation is used to specify that the value of a bean property is required to be dependency injected. That means, an error is caused if a value is not specified for that property
- **@Autowired:** Prior to Spring 2.5, autowiring could be configured for a number of different approaches: constructor, setters by type, setters by name, or autodetect – which offer a large degree of flexibility, but not very fine-grained control. For eg, it has not been possible to autowire a specific subset of an object's setter methods or to autowire some of its properties by type and others by name.

By using @Autowired, you can eliminate the additional XML in your configuration file that specifies the relationship between two objects. Also, you no longer need methods to set the property in the owning class. Just include a private or protected variable and Spring will do the rest.

- **@Resource :** Declares a reference to a resource such as a data source, Java Messaging Service (JMS) destination, or environment entry. This annotation is equivalent to declaring a resource-ref, message-destination-ref, env-ref, or resource-env-ref element in the deployment descriptor.

- **@PostConstruct** : Specifies a method that the container will invoke after resource injection is complete but before any of the component's life-cycle methods are called.
- **@PreDestroy** : Specifies a method that the container will invoke before removing the component from service.

When Spring discovers one of these annotations, it creates the appropriate bean by matching the stereotype.

Annotations to configure beans:

@Component: Is the basic stereotype. Classes annotated with this will become spring beans.

@Controller: Classes annotated with this annotation will be considered as a Controller in Spring MVC support.

@Repository: Classes with @Repository annotation represent a repository (eg a data access object)

@Service: This annotation marks classes that implement a part of the business logic of the application.

Some other annotations:

@Transactional: is an alternative to the XML-based declarative approach to transaction configuration. All methods of a Spring bean instantiated from a class with the @Transactional annotation will be transactional (thus indirectly, all methods will execute in a transaction). The functionality offered by the @Transactional annotation and the support classes is only available to you if you are using at least Java 5

@Aspect : This annotation on a class marks it as an aspect along with @Pointcut definitions and advice (@Before, @After, @Around)

We shall be covering each of these as we move forwards into AOP, database programming and web applications.

Notice that in all the XML configurations so far, we have used the beans namespace only. But the core Spring Framework comes with ten configuration namespaces as shown in next slide.



XML namespaces through which you can configure the Spring container

Name	Purpose
aop	Provides elements for declaring aspects and for automatically proxying @AspectJ annotated classes as Spring aspects.
beans	The core primitive Spring namespace, enabling declaration of beans and how they should be wired.
context	Comes with elements for configuring the Spring application context, including the ability to autodetect and autowire beans and injection of objects not directly managed by Spring.
jee	Offers integration with Java EE APIs such as JNDI and EJB.
jms	Provides configuration elements for declaring message-driven POJOs.
lang	Enables declaration of beans that are implemented as Groovy, JRuby, or BeanShell scripts.
mvc	Enables Spring MVC capabilities such as annotation-oriented controllers, view controllers, and interceptors.
oxm	Supports configuration of Spring's object-to-XML mapping facilities.
tx	Provides for declarative transaction configuration.
util	A miscellaneous selection of utility elements. Includes the ability to declare collections as beans and support for property placeholder elements.

1.6: Spring Annotations @Autowired annotation



```
<?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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config />

    <context:component-scan base-package="training.spring" />

    <!-- bean declarations go here -->
</beans>
```



- 54 -

We can use annotations to automatically *wire* bean properties. It is similar to `autowire` attribute in configuration file, but allows more fine-grained autowiring, where you can selectively annotate certain properties for autowiring. However, simply annotating your classes is not enough to get an annotation's behavior. You need to enable a component that is aware of the annotation and that can process it appropriately. This component (a special BeanPostProcessor implementation) will be different for all annotations. For eg, the `RequiredAnnotationBeanPostProcessor` class is necessary for `@Required` annotation.

Annotation wiring isn't turned on in the Spring container by default. So, before we can use annotation-based autowiring, we'll need to enable it in our Spring configuration. The simplest way to do that is with the `<context:annotation-config>` element from Spring's context configuration namespace.

`<context:annotation-config>` tells Spring that you intend to use annotation-based wiring in Spring. Once it's in place you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. The implicitly registered post-processors include `AutowiredAnnotationBeanPostProcessor`, `CommonAnnotationBeanPostProcessor`, `PersistenceAnnotationBeanPostProcessor`, as well as the `RequiredAnnotationBeanPostProcessor`.

So far, we have seen how `<context:annotation-config>` eliminates most uses of `<property>` and `<constructor-arg>` elements from your Spring configuration. But we still need to explicitly declare beans using `<bean>`.

The `<context:component-scan>` element is similar to `<context:annotation-config>`. But it also configures Spring to automatically discover beans and declare them, that is, most of the beans in your Spring application can be declared and wired without using `<bean>`.

The `<context:component-scan>` element scans a package and all of its subpackages, looking for classes that could be automatically registered as beans in the Spring container. The `base-package` attribute tells `<context:component-scan>` package to start its scan.

How does this work? By default, `<context:component-scan>` looks for classes that are annotated with one of the following: `@Component`, `@Controller`, `@Repository`, `@Service`.

For example, suppose that our application context only has the `currencyConverter` and `exchangeService` beans in it. We can eliminate the explicit `<bean>` declarations from the XML configuration by using `<context:component-scan>` and annotating the `CurrencyConverterImpl` and `ExchangeServiceImpl` classes with `@Component`.

1.6: Spring Annotations

DemoSpring_Aanno



- This demo illustrates autowired annotation



- 56 -

Please refer to **demo, DemoSpring_Aanno**, This example uses the **anno.xml** for configuring the beans.

Notice how the CurrencyConverterImpl.java depends on the com.igate.anno.ExchangeServiceImpl.java class for services. Notice how the exchangeService property has been annotated with @Autowired. The configuration file now just contains beans declaration and no instructions on wiring!

1.6: Spring Annotations

Annotating beans for autodiscovery



- Refer to demos, DemoSpring_Aanno



- 57 -

Refer to **demo**, DemoSpring_Aanno , Notice the anno.xml configuration file. It contains no beans! Notice the bean classes themselves are annotated with @Component, @PostConstruct, @Autowired etc.

Lab



➤ From the lab guide

- Lab-1 problem-statement-1 2 and 3



- 58 -

Lesson Summary



➤ We have so far seen:

- What is Spring and why spring?
- The Spring architecture
- Inversion of control
- Bean containers
- Lifecycle of beans in containers.
- Some popular implementations of BeanFactoryPostProcessors



- 59 -

Thus we have seen that Spring is the most popular and comprehensive of the lightweight J2EE frameworks that have gained popularity since 2003. We saw how Spring is designed to promote architectural good practice. A typical spring architecture will be based on programming to interfaces rather than classes.

We have seen what is Inversion of control and dependency injection. We also saw Bean containers and lifecycle of beans in containers. We saw how to hook into the lifecycle of a bean and make it aware of the Spring environment.

Review Questions



- **Question 1: The <constructor-arg> element has an optional _____ attribute that specifies the ordering of the constructor arguments.**
- Option 1: By index
 - Option 2: By type
 - Option 3: By order
- **Question 2: A _____ bean lets the container return a new instance each time a bean is asked for in a non-web application**
- Option 1: Singleton
 - Option 2: Prototype
 - Option 3: Request
 - Option 4: session



- 60 -

Review Questions



➤ **Question 3: Specifying the _____ tag will allow Spring to validate at deployment time that the other bean actually exists.**

- Option 1: idref
- Option 2: ref
- Option 3: local



➤ **Question 4: The BeanPostProcessor performs post processing on the entire Spring container.**

- Option 1: True
- Option 2: false

- 61 -





Spring Framework 3.0

Lesson 2: Spring Expression Language (SpEL)

Lesson Objectives



- **Introduction to SpEL (Spring Expression Language)**
 - SpEL Expression fundamentals
 - Expression Language features
 - Reduce configuration with @Value

- 2 -



2.1 : SpEL Expression fundamentals

What is SpEL?



- **The Spring Expression Language is a powerful expression language that supports querying and manipulating an object graph at runtime.**
- **SpEL supports many functionalities including:**
 - Literal expressions
 - Boolean and relational operators
 - Regular and class expressions
 - Accessing properties, arrays, lists, maps
 - Method invocation
 - Calling constructors
 - Bean references
 - Array construction
 - Inline lists
 - User defined functions
 - Templatized expressions

+ 3 +



So far we have seen how to wire dependencies using setter and constructor injections. But these have been statically defined in the Spring configuration file. When we wired the exchangeService property into the CurrencyConverter bean, that value was determined at development time. Likewise, when we wired references to other beans, those references were also statically determined in the Spring configuration.

What if we want to wire properties with values at runtime? Spring 3's *Spring Expression Language (SpEL)* is a powerful way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime. SpEL syntax is similar to Unified EL but offers additional features like method invocation and basic string templating functionality.

SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise. It thus can be used independently. It supports many functionalities as listed above.

2.2 : Expression Language features
Exploring literals and Types



➤ Working with Literals:

- Wire SpEL expression into a bean's property by using `#{ <exprn-string> }`

```
<property name="count" value="#{5}"/>
<property name="message" value="The value is #{5}"/>
<property name="frequency" value="#{89.7}"/>
<property name="name" value="#{'Chuck'}"/>
```

examples

➤ Working With Types:

- Use the T() operator to work with class-scoped methods & constants
- Example: `T(java.lang.Math)` //expresses Java's Math class in SpEL

```
<property name="multiplier" value="#{T(java.lang.Math.PI)}"/>
<property name="randomNumber" value="#{T(java.lang.Math.random())}"/>
```

examples



4

Literal Values: SpEL expressions, like any other expression, are evaluated for some value. SpEL can evaluate literal values, references to a bean's properties, a constant on some class etc. The simplest SpEL expression for example is 5, which evaluates to an integer value of 5. We can wire this value into a bean's property by using `#{ }` markers in a `<property>` element's value attribute, as shown in example above. The `#{ }` markers indicate that the content that they contain is a SpEL expression.

Similarly, see example above for expressing Floating-point numbers. Literal String values can be expressed in SpEL with either single or double quote marks. You can also use Boolean true and false values. Eg- `<property name="enabled" value="#{false}"/>`

Working With Types: The result of the T() operator is a Class object that represents the given class. The T() operator thus gives us access to static methods and constants on a given class. Eg, to wire the value of `pi` into a bean property, use:

```
<property name="multiplier" value="#{T(java.lang.Math.PI)}"/>
```

Likewise, static methods can also be invoked on the result of the T() operator. Eg, to wire a random number (between 0 and 1) into a bean property, use:

```
<property name="randomNumber" value="#{T(java.lang.Math.random())}"/>
```

When the application is starting up and Spring is wiring the randomNumber property, it'll use the `Math.random()` method to determine a value for that property!

2.2 : Expression Language features

Referencing Beans, Properties, And Methods

- SpEL allows to wire one bean into another bean's property by using the bean ID as the SpEL expression:

```
<property name="exchangeService" value="#{exchangeService}">
```

```
<bean id="currencyConverter" class="training.CurrencyConverterImpl">
    <property name="exchangeRate"
        value="#{exchangeService.exchangeRate}" />
</bean>
```

```
<property name="exchangeService"
    value="#{exchangeService.getExchangeRate()}">
```

Bean ID

Property name

referencing beans properties

Invoking methods

A SpEL expression can reference another bean by its ID. See first example above. We used SpEL to wire the bean whose ID is "exchangeService" into an exchangeService property. We can do this by using the ref attribute too! <property name="exchangeService" ref="exchangeService"/> The outcome is the same. But let us see how to take advantage of being able to wire bean references with SpEL. The second example configures a new CurrencyConverterImpl bean whose ID is currencyConverter. This is wired to whatever exchangeRate the exchangeService bean provides. This is equivalent to:

```
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl ();
currencyConverter.setExchangeRate(exchangeService.getExchangeRate());
```

2.2 : Expression Language features

Performing operations on SpEL values

- SpEL includes several operators to manipulate the values of an expression.

examples

Operation type	Operators
Arithmetic	+, -, *, /, %, ^
Relational	<, >, ==, <=, >=, lt, gt, eq, le, ge
Logical	and, or, not,
Conditional	? : (ternary), ?: (Elvis)
Regular expression	matches

```

<property name="adjustedAmount" value="#{counter.total + 42}"/>
<property name="result" value="#{2 * T(java.lang.Math).PI * circle.radius}"/>
<property name="fullName" value="#{emp.firstName + ' ' + emp.lastName}"/>
<property name="redCustomer" value="#{account.balance le 100000}"/>
<property name="outOfStock" value="#{!product.available}"/>
<property name="outcome"
    value="#{T(java.lang.Math).random() > .5 ? 'win' : 'lose'}" />

```



- 6 -

Like in Java, + operator is overloaded to perform concatenation on String values. We know that the less-than (<) and greater-than (>) operators are used to compare different values. Unfortunately, they pose a problem when using these expressions in Spring's XML configuration (since they have special meaning in XML). So, when using SpEL in XML, it's best to use SpEL's textual alternatives like le (<), gt (>) etc.

SpEL supports regular expressions using the *matches* operator, which is a relational operator returning true or false. Example:

```

<util:map id="regExpsSamples" value-type="java.lang.Object"
          key-type="java.lang.String">
    <entry key="# {'a string' matches 'a.*'}"
          value="#{'a string' matches 'b.*'}"/>
</util:map>

```

2.2 : Expression Language features
Working with collections



```
package trg.spring;
public class City {
    private String name;
    private String state;
    private int population;
    //setter and getter methods for each of these properties
}
```

```
<util:list id="cities">
    <bean class="trg.spring.City"
        p:name="Chicago" p:state="IL" p:population="2853114"/>
    <bean class="trg.spring.City"
        p:name="Atlanta" p:state="GA" p:population="537958"/>
    <bean class="trg.spring.City"
        p:name="Dallas" p:state="TX" p:population="1279910"/>
    .....
</util:list>
```



Let us digress a bit and see how Spring allows us to create collection via configuration file.

From Spring 2.5 onwards, in addition to standard `<property>` tag, you can use the `p` namespace to set properties of beans.

The `<util:list>` element comes from Spring's util namespace. It creates a bean of type `java.util.List` that contains all of the values or beans that it contains. In this case, that's a list of City beans.

Example for creating Map:

```
<util:map id="numbersMap" value-type="java.lang.Integer"
    key-type="java.lang.String">
    <entry key="one" value="1"/>
    <entry key="two" value="2"/>
    <entry key="three" value="3"/>
    <entry key="four" value="4"/>
    <entry key="five" value="5"/>
</util:map>
```

2.2 : Expression Language features

Accessing collections



- 1 <property name="customerCity" value="#{cities[2]}"/>
- 2 <property name="customerCity" value="#{cities['Dallas']}"/>
- 3 <property name="userName" value="#{userprops['user.name']}"/>
- 4 <property name="smallCities" value="#{cities.? [population lt 100000]}"/>
- 5 <property name="cityNames" value="#{cities.! [name]}"/> selection
- 6 <property name="cityNames" value="#{cities.! [name + ', ' + state]}"/> projection
- 7 <property name="cityNames" value="#{cities.? [population gt 100000].! [name + ', ' + state]}"/>



Example -1 : This selects the third city from the list to assign to customerCity

Example -2: Assuming that cities is a java.util.Map collection, with city-name as the key. The example shows how to retrieve the entry for Dallas.

Example -3: We can use the [] operator is to retrieve a value from a java.util.Properties collection too. First: load a properties configuration file into Spring using the <util:properties> element as follows:

```
<util:properties id="userprops" location="classpath:user.properties"/>
```

The userprops bean is a java.util.Properties that contains all of the entries in the file named user.properties. Accessing a property from that file is similar to accessing a member of a Map. The 3rd example above reads a property whose name is user.name from the userprops bean

Example -4: We would like a list of cities whose population is less than 100,000. Use the selection operator (.?[]) when doing the wiring. The selection operator creates a new collection whose members include only those members from the original collection that meet the criteria expressed between the square braces. In this case, the smallCities property will be wired with a list of City objects whose population property is less than 100,000.

Example -5: Projecting collections means collecting a particular property from each of the members of a collection into a new collection. Use SpEL's projection operator (.![]) to do this. Example-5 retrieves a list of city names from the collection of City objects. You can also retrieve multiple members as the next example shows.

The final example (7) is a combination of selection and projection.

2.3 : Reduce configuration with @Value
@Value annotation



```
package training.spring.spel;  
@Component("user")  
public class UserBean {  
    @Value("#{userprops.username}")  
    private String username;  
    @Value("#{userprops.password}")  
    private String password;  
    // setter and getter methods for properties
```

inject values from the properties files using a SpEL expression

```
<beans xmlns="http://www.springframework.org/schema/beans"  
.....>  
    <context:component-scan base-package="training.spring.spel" />  
    <util:properties id="userprops" location="classpath:user.properties" />  
</beans>
```

properties file



SpEL combined with @Value annotation is great. We have already seen how component scan and autowiring reduces the size of XML configuration files. However, we still have to deal with beans that need literal values as input. With @Value you can inject values from your properties files using SpEL. Assume that you have a properties file configured as shown above.

To use SpEL in annotation, you must register your component via annotation. If you register your bean in XML and define @Value in Java class, the @Value will fail to execute.



Demo : SpringDemo_SPEL



➤ Using SpEL



- 10 -

Please refer to **demo, SpringDemo_SPEL**

Lesson Summary



➤ We have so far seen:

- SpEL Expression fundamentals
- Expression Language features
- Reduce configuration with @Value



- 11 -

Review Questions



➤ **Question 1:** _____ markers indicate that the content that they contain is a SpEL expression.

- Option 1: \${ }
- Option 2: #{ }
- Option 3: %{ }



➤ **Question 2:** SpEL can access instances of `java.lang.Class` using the _____ operator

- Option 1: #{ }
- Option 2: T
- Option 3: Type



- 12 -

Review Questions



- **Question 3: The _____ effectively creates a bean of type `java.util.Map` that contains all of the values or beans that it contains .**
 - Option 1: `<util:list>`
 - Option 2: `<util:properties>`
 - Option 3: `<util:map>`

- **Question 4: If `@Value` annotation is used in a component, it is mandatory that the component be annotated with `@Component`**
 - Option 1: True
 - Option 2: False



- 13 -



Spring Framework 3.0

Lesson 3: JDBC in Spring



Lesson Objectives



➤ Introduction to Spring JDBC framework

- Learn core functionality of Spring's JDBC framework
- Understand templates and callback mechanism
- Understand How Spring provides a way to actually model database operations as objects
- Exception Handling



Topics covered:

Data access using JDBC

 JdbcTemplate and DataSource

 SimpleJdbcTemplate class

 NamedParameterJdbcTemplate class

 Executing statements

 Updating the database

Modeling JDBC operations as Java objects using

 SqlQuery , MappingSqlQuery , SqlUpdate

3.1: Spring JDBC
Introduction



- **DAO support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way**
- **Spring JDBC framework is built on top of the core JDBC API.**
 - Its main aim is to leverage the features of JDBC that work well and to abstract away all problem areas.
- **Spring provides a consistent exception hierarchy that is used across all of its DAO frameworks**



Introduction to Spring JDBC API :

The main process of any enterprise application is data persistence. Spring comes with a family of data access frameworks that integrate well with a variety of data access technologies.

Whether you are persisting your data via direct JDBC, Java Data Objects(JDO) or an object/relational mapping (ORM) tool like Hibernate, Spring separates persistence code from general business logic using DAO (Data Access Object) pattern.

The Spring JDBC framework is built on top of the core JDBC API. Its main aim is to leverage the features of JDBC that work well and to abstract away all problem areas. This focuses primarily around connection and exception handling, which is done by providing an abstraction layer on top of JDBC. This abstract layer manages connections as well as translates SQLException into more meaningful exceptions.

Spring helps you insulate your data access tier from the rest of the application by providing a consistent exception hierarchy that is used across all of its DAO frameworks. Till now, in JDBC code, you are forced to catch the SQLException which is often treated as “one size fits all” exception for problems related to data access. Spring on the other hand provides a rich exception hierarchy which is rooted in DataAccessException. Each of these exceptions are unchecked. This leaves the decision of whether to catch an exception in the developers hands.

3.1: Spring JDBC

Problem with JDBC code



```

public void GettingRows() {
    Connection conn=null;
    Statement stmt=null;
    Resultset rset=null;
    try{
        conn = dataSource.getConnection();
        stmt = conn.createStatement ();
        rset = stmt.executeQuery ("select empno, ename, job from emp");
        while (rset.next()) {
            System.out.print (rset.getString (1));
        }
    } catch (SQLException e) {
        LOGGER.error(e);
        throw e;
    }
    finally{
        //code to clean up resources
    }
}

```

Diagram illustrating the steps in the JDBC code:

- Declare connection parameters
- Open connection
- Create statement
- Execute statement
- Iterate over resultset
- Handle exceptions
- Clean up resources



- 4 -

The added value provided by the Spring Framework's JDBC abstraction framework is perhaps best shown by the following list (note that only the italicized lines need to be coded by an application developer):

1. Define connection parameters
2. Open the connection
3. *Specify the statement*
4. Prepare and execute the statement
5. Set up the loop to iterate through the results (if any)
6. *Do the work for each iteration*
7. Process any exception
8. Handle transactions
9. Close the connection

While JDBC gives you an API to work closely with your db, you are responsible for handling everything related to accessing the database. This includes managing db resources and handling exceptions. In the code above, the only lines that are critical are creating the statement and iterate over resultset. The rest is simply **boiler plate code**. But this boilerplate code is important. Eg cleaning up code and handling exceptions is what makes a code robust. Therefore, we need a framework where this code is written once & written right!

The Spring Framework takes care of all the grungy, low-level details that can make JDBC such a tedious API to develop.

Please refer to demo, ex-11, JDBCWithoutSpring.java in the “training.spring” package.

3.1: Spring JDBC

Using JDBC with Spring



➤ **Spring separates the fixed and variant parts of the data access process into two distinct classes:**

- Templates: manage the fixed part of the process like controlling transactions, managing resources, handling exceptions etc
- Callbacks: define implementation details, specific to application such as creating statements, binding parameters etc.

- 5 -



JDBC is still a very popular method of persisting data. However, with JDBC, an user is responsible for handling everything related to accessing data from database, right from obtaining connection, creating statement object to querying the database, clean up resources after a database operation etc.

Data access can be compared to, say, the postal system. Most of the steps are more or less consistent. We post a letter, it travels via some transportation mode, arrives at a destination, is classified and comes into your mailbox. You are involved only when you need to be; the rest is just "taken care of". This mirrors a very powerful design pattern – the template method pattern and defines the fixed part of any process.

But at some places, the steps may delegate to other processes. Eg validating is different for different letters or parcels. This is the part of the process that is variable. In software terms, a template method delegates the implementation specific portions of code to an interface.

In a typical JDBC application, we always need to obtain a connection and clean up resource after a database operation. These are *fixed steps*, also called boilerplate code. However, some code might query DB whereas others might update. These are *variable steps*.

Spring separates the fixed and variant parts of the data access process into two distinct classes: *templates and callbacks*. Templates manage the fixed part of the process. Callbacks define implementation details, specific to your application.

3.2: Spring's Template classes

Data Access Templates

 iGATE
ITOPS for Business Outcomes

➤ **Spring offers several data access templates, each suitable for a different persistent mechanism. Some of them are listed below:**

Template class:	Suitable for:
jdbc.core.JdbcTemplate	JDBC connections
jdbc.core.simple.SimpleJdbcTemplate	JDBC connections, simplified with Java 5 constructs
Jdbc.core.namedparam.NamedParameterJdbcTemplate	JDBC connections with support for named parameters
orm.hibernate.HibernateTemplate	Hibernate 2.x sessions
orm.jdo.JdoTemplate	Java data objects implementations
orm.jpa.JpaTemplate	Java persistent API entity managers

- 6 -



Spring comes with several templates to choose from, depending on your persistence platform choice. If you are using straight JDBC, you will want to use **JdbcTemplate**. But, if you are using an ORM framework, then **HibernateTemplate** or **JpaTemplate** is more appropriate.

Using a data access template simply involves configuring it as a bean in your Spring context and then wiring it in your application DAO. You can also take advantage of Spring's DAO support classes to further simplify your application DAOs.

For JDBC, Spring comes with three template classes to abstract boilerplate data access code:

- **JdbcTemplate** – is the most basic of JDBC templates. Provides simple access to databases through JDBC and simple indexed-parameter queries.
- **NamedParameterJdbcTemplate** – allows to execute queries where values are bound to named parameters in SQL, rather than indexed parameters (traditional JDBC "?" place holders).
- **SimpleJdbcTemplate** - Java-5-based convenience wrapper for the classic Spring **JdbcTemplate**, taking advantage of varargs and autoboxing, and exposing only the most commonly required operations in order to simplify **JdbcTemplate** usage.

You may refer to appendix-A to understand **NamedParameterJdbcTemplate** and **SimpleJdbcTemplate** better.

3.2: Spring's Template classes
The JdbcTemplate class



➤ **Central class in JDBC framework**

- Manages all database communication and exception handling
- Based on template style of programming; some calls are handled entirely by JdbcTemplate while others require the calling class to provide callback methods that contain implementation for parts of the JDBC workflow

- 7 -



At the core of the Spring JDBC support is the JdbcTemplate class. This class allows you to work directly with instances of Statement, PreparedStatement and CallableStatement classes. The JdbcTemplate helps manage the connection and translates any SQLException to a Spring data access exception.

This class is based on template style of programming; some calls are handled entirely by jdbc template while others require the calling class to provide callback methods that contain implementation for parts of the jdbc workflow. All that JdbcTemplate needs is the data source to create an instance which is taken up in the next page.

JdbcTemplate executes the core JDBC workflow like statement creation and execution, leaving 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. In a nutshell, the JdbcTemplate offers a plethora of convenience methods for queries, inserts, updates and executions; it lets you focus on writing the SQL and lets Spring deal with ugly JDBC code.

We shall be covering the jdbc operations that JdbcTemplate can perform later in this lesson.

3.3: Configuring data sources declaratively

The DataSource



- A **DataSource** obtains a connection to the database for working with data.
- Some of the implementations of **DataSource** are:
 - BasicDataSource
 - PoolingDataSource
 - SingleConnectionDataSource
 - DriverManagerDataSource
- The datasource can be programmatically configured, for example:

```
DriverManagerDataSource ds = new DriverManagerDataSource();
ds.setDriverClassName(driver);
ds.setUrl(url);
ds.setUsername(username);
ds.setPassword(password);
```

- 8 -



To work with data from a database, we need a **DataSource**. A **DataSource** is a JDBC interface that acts as a factory for connections. Since it's an interface, it has many implementations. The most commonly used DataSources are `org.apache.commons.dbcp.BasicDataSource` and `org.apache.commons.dbcp.PoolingDataSource`.

You can also use a **DataSource** implementation provided by Spring. The `org.springframework.jdbc.datasource.DriverManagerDataSource` implementation will create a new connection each time the framework calls the `getConnection()` method. This works well for unit testing, but not a good choice if application makes multiple calls to the database.

For applications where you want to acquire the connection only once and hold on to it, Spring provides the `org.springframework.jdbc.datasource.SingleConnectionDataSource`. With this implementation, you should call the `destroy()` method when you are done with database processing.

3.3: Configuring data sources declaratively

Example

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

<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="/jdbc/TrgDatasource"/>
    <property name="resourceRef" value="true"/>
</bean>
```



- 9 -

The datasource can also be configured declaratively. For your data access objects, you normally declare a `DataSource` variable and set it using dependency injection.

We can obtain a datasource from JNDI, from a data source connection pool or use a `DriverManagerDataSource` class as seen in the code.

Some examples of defining the datasource bean are shown above.

In the second example, we are depending upon an external datasource configuration which will be obtained via a JNDI lookup. The `JndiObjectFactoryBean` looks up data source and injects into Spring application.

3.3: Configuring data sources declaratively
Wiring beans in the Spring Context file



```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
    <property name="dataSource"><ref local="myDataSource" /></property>
  </bean>
  <bean id="myDataSource" ....
        <!-- the datasource configuration comes here -- >
  </bean>
  <bean id="jdbcTemplateDemo" class="JdbcTemplateDemo">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
  </bean>
```

```
class JdbcTemplateDemo{
  SimpleJdbcTemplate jdbcTemplate;
  public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate){
    this.jdbcTemplate= jdbcTemplate;
  }}
```

- 10 -



In the above example, we have wired the datasource bean to the SimpleJdbcTemplate bean. The SimpleJdbcTemplate bean is then injected into the application bean (JdbcTemplateDemo). Alternatively, you can simply configure the datasource in the context file and then inject it into the application. In this case, you will need to programmatically create the JdbcTemplate instance in this manner:

```
<bean id="jdbcTemplateDemo" class="JdbcTemplateDemo">
  <property name="dataSource" ref="dataSource" />
</bean>
```

```
class JdbcTemplateDemo{
  SimpleJdbcTemplate jdbcTemplate;
  ....
  jdbcTemplate= new SimpleJdbcTemplate(ds);
  public void set DataSource(DataSource dataSource){
    this.ds = dataSource;
  }}
```

3.4: Flavors of JdbcTemplate

Query and update methods: Examples

```

    > JdbcTemplate jt = new JdbcTemplate(dataSource);

    int count = jt.queryForInt("select count(*) from emp");
    String name = (String) jt.queryForObject("select name from mytable where empno=1022", String.class);
    List rows = jt.queryForList("select * from mytable");
    Object params[] = new Object[]{new Double(1000.0)};
    List rows1 = jt.queryForList("Select * from emp where sal > ?",params);

    int x=jt.update("insert into Book(id,name) values(1,'Core Java')");
    int x=jt.update("Update Book set name='Advanced Java' where id=?", new Object[]{new Integer(3)});
    int x=jt.update("Delete from Book where id=2");

    String sql = "insert into person(id,fname,lname) values (?,?,?)";
    Object[] params = new Object[]{p.getId(), p.getFname(), p.getLname()};
    int count = jdbcTemplate.update(sql,params);
  
```

- 11 -

LEARN
iGATE Learning Education and Research Node

There are several convenience methods in JdbcTemplate that provide an easy interface for accessing and manipulating data in the DB. (Please refer to the JdbcTemplate class in the Java documentation for more details). These convenience methods provide basic options for executing queries and updating database.

We have query methods with different variations – you can either pass static sql or you can pass in sql with placeholders for your parameters along with an object array containing the values for your parameters.

Some of the convenient methods provided by JdbcTemplate class are :

1. `queryForInt()`, `queryForLong()` or `queryForObject()` : use these methods if you want to retrieve a count or a specific value from one row. In the first example, the count variable will contain the number of employees, since the `queryForInt` returns a single int returned by the sql statement.
2. `queryForObject()` : will convert the returned JDBC Type to the Java class that is passed in as an argument. If the type conversion is invalid, then an Exception will be thrown.
3. `queryForList()` : returns a List with an entry for each row that the query returned. The fourth example returns a list based on a prepared statement. Here params is an array containing arguments for each of the parameters (here only one). After replacing the parameters, query is executed to return a list.

update methods: We can simply pass in a SQL statement or if we use parameter placeholders in SQL, we can also pass in an Object array with the actual parameters.

Please refer to the examples given above.

3.4: Flavors of JdbcTemplate
JdbcTemplate : execute methods



```
public class ExecuteAStatement {  
    private JdbcTemplate jt, DataSource dataSource;  
    public void doExecute() {  
        jt = new JdbcTemplate(dataSource);  
        jt.execute("create table mytable (id integer, name varchar(100))");  
    }  
    public void setDataSource(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
}
```



- 12 -

execute methods: In addition to query methods, there are overloaded execute() methods. Some of them are :

- Object execute(ConnectionCallback) : performs the code in ConnectionCallback, doInConnection() and returns the value returned from the callback
- Object execute(StatementCallback) : performs the callback in StatementCallback, doInStatement(). Use this operation to perform operations on a single statement obtained from the connection.
- void execute(String) : Simplest; performs a single statement using the String argument supplied

The above code example (class ExecuteAStatement) shows code to include for a minimal but fully functional class that creates a new table. The doExecute() method creates a new instance of the JdbcTemplate. The datasource has been provided by invoking the setDataSource method prior to calling the executeAstatement method. The convinience method of the JdbcTemplate class execute executes the DDL statement to create a table.

Demo :Using JDBC Template Class



- Execute the :
 - DemoJDBC



- 13 -

Please refer to demo, DemoJDBC, EmployeeDao.java

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case you annotate the class with `@Repository` (which makes it a candidate for component-scanning) and annotate the `SimpleJdbcTemplate` setter method with `@Autowired`.

3.5: Spring's DAO Support

JdbcDaoSupport class

iGATE
ITOPS for Business Outcomes

- On top of the template-callback design, Spring provides DAO support classes that are meant to be subclassed.
 - When writing your application DAO implementation, you can call a template retrieval method to have direct access to the underlying data access template.
 - To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend

```

graph LR
    DAO[DAO support] --> Template[Data access template]
    Template --> Persistence[Persistance framework]
    Persistence --> Database[database]
    DAO <--> ApplicationDAO[Application DAO]
  
```

- 14 -

LEARN
iGATE Learning, Education and Research Node

There is more to Spring's data access framework than just Data access templates. On top of the template-callback design, Spring provides DAO support classes that are meant to be subclassed. The figure above illustrates the relationship between a template class, a DAO support class, and a custom DAO implementation.

DAO support classes provide convenient access to the template class that they support. When implementing DAO, you can call a template retrieval method to have direct access to the underlying data access template.

Example, if your application DAO subclasses JdbcDaoSupport, then call getJdbcTemplate() to get a JdbcTemplate to work with, call getDataSource() to work with DataSource!

Why “support” classes? You could use all the JDBC support classes we have introduced in any class so far. However, it is very likely that you will use them only in your data access tier. Therefore Spring comes with the convenience class JdbcDaoSupport. This is a convenient super class for JDBC-based data access objects. It requires a DataSource to be set, providing a JdbcTemplate based on it to subclass through the getJdbcTemplate() method.

To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend. These abstract classes have methods for setting the data source and any other configuration settings that are specific to the technology you currently are using. They are JdbcDaoSupport, HibernateDaoSupport , JdoDaoSupport, JpaDaoSupport etc. We shall see more on these in later modules.

3.5: Spring's DAO Support
DemoJDBCDao



- This demo shows how to use the JdbcDaoSupport class.



- 15 -

The Employee.java is a POJO. The EmployeeDAO extends the JdbcDAOsupport class. This class gives you the getJdbcTemplate() method that returns an instance of the jdbc template class.

The getEmployeeList() method creates a new instance of the RowMapper interface. The row mapper interface is responsible for mapping a resultset row to an object. So the inner class is invoked for each Row on the ResultSet. It implements the Spring RowMapper interface, that is used to convert ResultSet records to Domain Objects – here employee objects.

Run the JdbcClient application. Notice that getBean() creates a new instance of data source (refer to jdbc.xml).

3.6: Exception Handling

Handling Exceptions in the Spring JDBC



➤ Data Access Exception:

The Spring framework offers a consistent data access exception-handling mechanism for its data access module, including the JDBC framework. In general, all exceptions thrown by the Spring JDBC framework are subclasses of `DataAccessException`, a type of `RuntimeException` that you are not forced to catch. It's the root exception class for all exceptions in Spring's data access module.

Ex 1:

```
try
{
    employeeDao.insert(employee);
}
catch (DuplicateKeyException e)
{
    System.out.println("Employee Already Exist");
} catch (DataAccessException e)
{
    // Code to handle exception
}
```

- 16 -



Why Data Access Exception?

Many of the JDBC APIs declare throwing `java.sql.SQLException`, a checked exception that must be caught. It's very troublesome to handle this kind of exception every time you perform a database operation. You often have to define your own policy to handle this kind of exception. Failure to do so may lead to inconsistent exception handling.

EX 2:

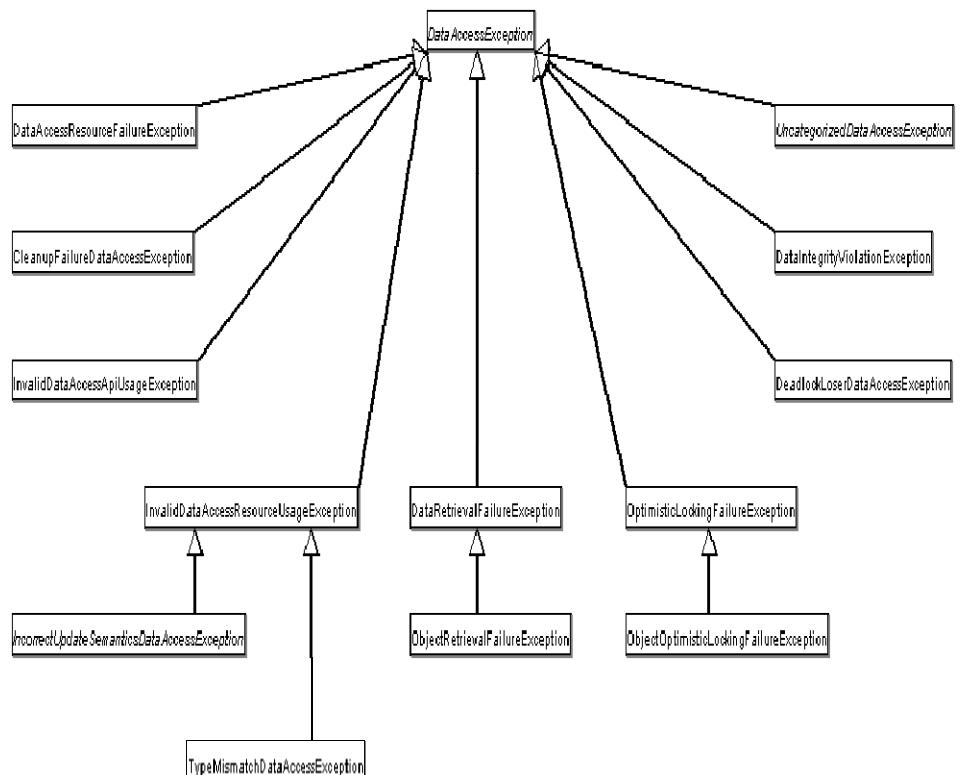
```
try {
    Employee
employee=employeeDao.getEmployee(2));
} catch (EmptyResultDataAccessException e) {
System.out.println("The Employee id is invalid");
} catch (DataAccessException e) {
// Code to handle exception
}
```

Cont...

JDBC does not offer an exception hierarchy, but throws SQLException in response to all errors. Finding out what actually went wrong - for example, was the problem a deadlock or invalid SQL? - involves examining the SQLState value and error code. The meaning of these values varies between databases.

Spring on the other hand , provides APIs that move tedious and error-prone exception handling out of application code into the framework. The framework takes care of all exception handling; application code can concentrate on issuing the appropriate SQL and extracting results. By providing a meaningful exception hierarchy for your application code to work with in place of SQLException. When Spring first obtains a connection from a DataSource it examines the metadata to determine the database product. It uses this knowledge to map SQLExceptions to the correct exception in its own hierarchy descended from org.springframework.dao.DataAccessException. Thus your code can work with meaningful exceptions, and need not worry about proprietary SQLState or error codes. Spring's data access exceptions are not JDBC-specific, so your DAOs are not necessarily tied to JDBC because of the exceptions they may throw.

The exception hierarchy that Spring provides can be seen below. (Please note that the class hierarchy detailed in the image shows only a subset of the entire DataAccessException hierarchy.)



3.6: Exception Handling

DemoDataAccessException



- This demo shows how to handle exception in spring based JDBC application



- 18 -

Understand how **DataAccessException** is handled in the application. Note that exception is handled in client instead of EmployeeDao class, When client invokes insertRec() and getEmployee() methods.

Lab



➤ Lab-2 from the lab guide



- 19 -

Lesson Summary



➤ We have so far seen:

- How Spring's JDBC framework offers an excellent way to implement very low-level data access code without having to spend too much time writing tedious JDBC code.
- Convenience methods of the JdbcTemplate class
- How DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way.



- 20 -



Thus we have seen how Spring's JDBC framework offers an excellent way to implement very low-level data access code without having to spend too much time writing tedious JDBC code. We saw how to use datasource to obtain connections. We also saw how to use JdbcTemplate to query and update data. We learnt about some convenience methods of the JdbcTemplate class and how Spring provides a way to actually model database operations as objects

The StoredProcedure class (see Appendix-E) is spring's sophisticated approach to simplify stored procedure invocation. There are more advanced issues that spring can handle:

Spring handles large objects such as BLOBs and CLOBs, using both JdbcTemplate and RdbmsOperations support. Spring also offers jdbc batching support where you can group a number of statements together and send them together to database for processing.

The DAO support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way. For data to be accessed in a consistent way, Spring provides a set of abstract DAO classes that you can extend. These abstract classes have methods for setting the data source and any other configuration settings that are specific to the technology you are using currently.

Review Questions

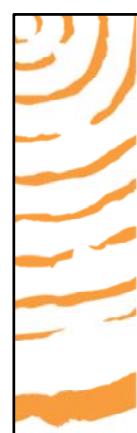


- **Question 1:** _____ classes manage the fixed part of the process like controlling transactions and managing resources.
 - Option 1: Template
 - Option 2: Pattern
 - Option 3: Callback
 - Option 4: Support

- **Question 2: DataAccessException is Checked Exception.**
 - Option 1: TRUE
 - Option 2: FALSE



- 21 -



Spring Framework 3.0

Lesson 4: Spring MVC framework

Lesson Objectives



➤ Introduction to Spring MVC framework

- Learn how to develop web applications using Spring
- Understand the Spring MVC architecture and the request cycle of Spring web applications
- Understand components like handler mappings, ViewResolvers and controllers
- Use MVC Annotations like @Controller, @RequestMapping and @RequestParam



+ 2 +

Web applications have become a very important part of any enterprise system. The key requirements for a web framework is to simplify development of the web tier as much as possible. Spring provides a web framework based on the MVC (Model view Controller) paradigm. Although it is similar in some ways to other popular MVC frameworks such as Struts and WebWork, Spring web MVC provides significant advantages over those frameworks.

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in separating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

4.1 : Spring MVC introduction

Spring MVC Framework Features

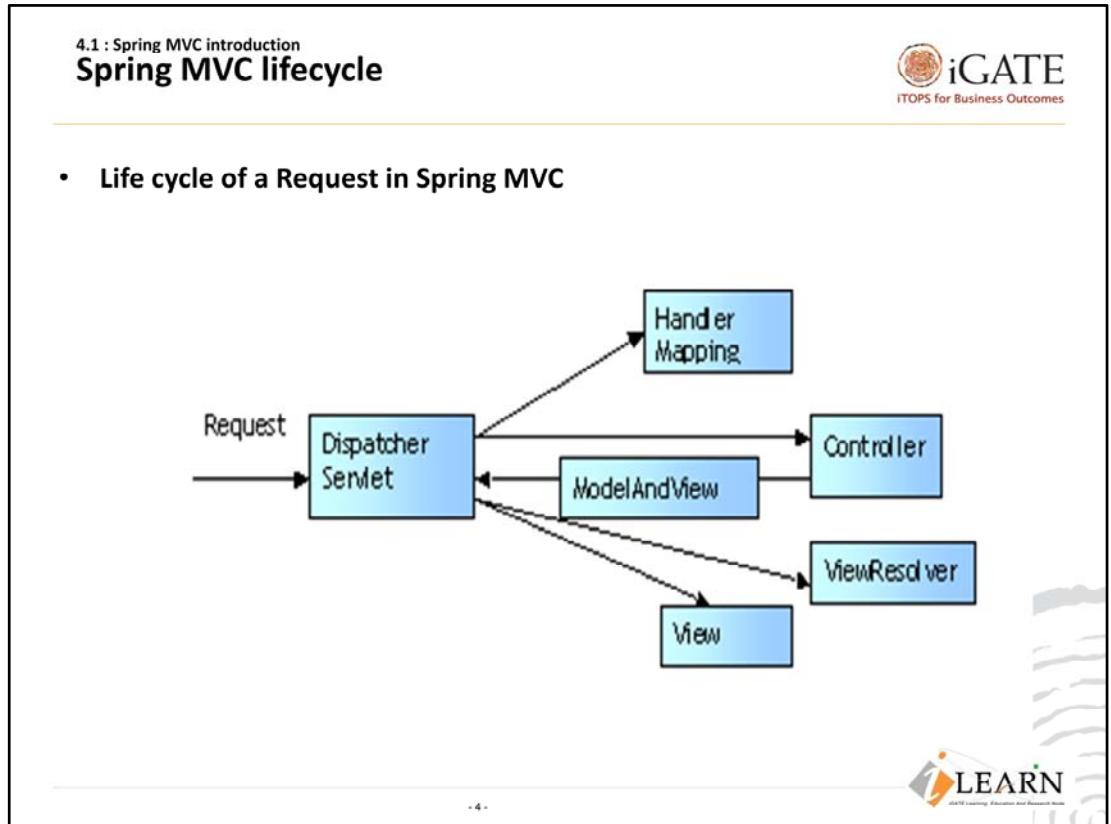
iGATE
ITOPS for Business Outcomes

- Provides you with an out-of-the-box implementations of workflow typical to web applications
- Allows you to use a variety of different view technologies
- Enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection
- Displays modular framework, with each set of components having specific roles and completely decoupled from the rest of the framework

LEARN
iGATE Learning, Education and Research Network

The Spring MVC is a web framework built within the Spring Framework. There are a number of challenges while creating a web-based application - like state management, workflow and validation, which are addressed by Spring's web framework. This framework can be used to automatically populate your model objects from incoming request parameters while providing validation and error handling as well. The entire framework is modular, with each set of components having specific roles and completely decoupled from the rest of the framework. This allows you to develop the front end of your web application in a very pluggable manner.

MVC provides out-of-the-box implementations of workflow typical to web applications. Its highly flexible, allowing you to use a variety of different view technologies. It also enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection. You can use Spring web MVC to make services that are created with other parts of Spring available to your users, by implementing web interfaces.



Life cycle of a request in Spring MVC: From the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring MVC framework are involved.

The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.

The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to Controller objects.

Once the DispatcherServlet has a Controller object, it dispatches the request to the Controller which performs the business logic (a well-designed Controller object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, the Controller returns a ModelAndView object to the DispatcherServlet. The ModelAndView object contains both the model data and logical name of view. The DispatcherServlet queries a ViewResolver with this logical name to help find the actual JSP. Finally the DispatcherServlet dispatches the request to the View object, which is responsible for rendering a response back to the client.

4.2 : Building a basic Spring MVC application

Configuring the DispatcherServlet in web.xml



The servlet-name given to the servlet is significant

```
<servlet>
  <servlet-name>basicspring</servlet-name>
  <servlet-class> org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>basicspring</servlet-name>
  <url-pattern>*.obj</url-pattern>
</servlet-mapping>
```

Steps to build a homepage in Spring MVC:

- Write the controller class that performs the logic behind the homepage
- Configure controller in the DispatcherServlet's context configuration file
- Configure a view resolver to tie the controller to the JSP
- Write the JSP that will render the homepage to the user



- 5 -

Configuring the DispatcherServlet

The dispatcher servlet is at the heart of the Spring MVC and functions as Spring MVC's front controller. Like any other servlet, it must be configured in web.xml file. Place the <servlet> declaration (in the first listing above) in the web.xml file.

The servlet-name given to the servlet is significant. By default, when DispatcherServlet is loaded, it will load the Spring application context from an xml file whose name is based on the name of the servlet. In the above example, the servlet-name is basicspring and so the DispatcherServlet will try to load the application context from a file called basicspring-servlet.xml.

Next, you must indicate which URL's will be handled by DispatcherServlet. Add the following <servlet-mapping> tag (the second xml listing) to web.xml to let DispatcherServlet handle all url's that end in .obj. The URL pattern is arbitrary and could be any extension.

DispatcherServlet is now configured and ready to dispatch requests to the web layer of your application.

Pls. see the above steps that are followed while building a simpleMVC application. These steps are explained in detail in the coming slides.

4.2 : Building a basic Spring MVC application
Breaking Up the Application Context

iGATE
ITOPS for Business Outcomes

- **Configuring the ContextLoaderListener in web.xml**

```
<listener>
<listener-class>
    org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

- **Setting ContextConfigLocation Parameter**

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
    /WEB-INF/basicsspring-service.xml
    /WEB-INF/basicsspring-data.xml
</param-value>
</context-param>
```

LEARN

We have seen that DispatcherServlet will load the Spring application context from a single XML file whose name is <servlet-name>-servlet.xml. But you can also split the application context across multiple XML files. Ideally splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. To ensure that all the configuration files are loaded, you will need to configure a context loader in your web.xml file. A context loader loads context configuration files in addition to the one that DispatcherServlet loads. The most commonly used context loader is a servlet listener called ContextLoaderListener that is configured in web.xml as shown in the first listing above.

With ContextLoaderListener configured, we need to tell it the location of the Spring configuration file(s) to load. If not specified, context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. We will specify one or more Spring configuration file(s) by setting the contextConfigLocation parameter in the servlet context as seen in the second listing above. The context loader will use contextConfigLocation to load the two context configuration files – one each for the service and data layer.

4.2 : Building a basic Spring MVC application

Building a Homepage in Spring MVC

 iGATE
ITOPS for Business Outcomes

➤ **Step 1 : Write the controller class that performs the logic**

- Controller handles request and performs business logic
- The handleRequest() returns a ModelAndView object

```
public class BasicSpringController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException{
        String now = new java.util.Date().toString();
        return new ModelAndView("hello","now",now);
    }
}
```

➤ **Step 2: Configure the controller in the DispatcherServlet's context configuration file**

```
<bean id="basicsspringController"
      class="training.web.spring.BasicSpringController" />
```

- 7 -



Step 1 : Write the controller class that performs the logic behind the homepage. The Controller is responsible for handling a request and performing business processing. It is almost similar to an HttpServlet or the Struts Action. However, the Spring MVC Controller is configured as just another JavaBean in the Spring application context.

The handleRequest() returns a ModelAndView object (See the first code listing above). This is an object that holds both view information and model data that will be used while rendering the output. In above example, it tells DispatcherServlet to take the user to the view whose name is **hello** and to place the **now** String object into the “now” field of the model data.

Step 2 : Configure the controller in the DispatcherServlet's context configuration file. By default, when DispatcherServlet is loaded, it will load the Spring application context from an xml file whose name is based on the name of the servlet. The Controller bean must be configured in the configuration file (basicsspring-servlet.xml) as seen in the second listing above.

Instead of specifying a bean id for the Controller bean, we can also use a name attribute.

Eg: <bean name="/hello.obj" class="training.spring.BasicSpringController" />
 It would then specify the name of bean and also the URL pattern for requests that should be handled by this controller. Because the URL pattern has special characters that are not available in an XML id attribute – specifically the slash (/) character – the name attribute must be used instead of id. This in this case, whenever a request comes to DispatcherServlet with a URL that ends with /hello.obj, DispatcherServlet will dispatch the request to BasicSpringController for handling.

4.2 : Building a basic Spring MVC application
Building a Homepage in Spring MVC (Contd..)

Step 3 : Configure a view resolver to bind controller to the JSP

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/</value></property>
    <property name="suffix"><value>.jsp</value></property>
  </bean>
```

} /hello.jsp

Step 4 : Write the JSP that will render the homepage to the user

```
<html>
  <body>
    <h1>Welcome to Spring!! </h1>
    Now it is ${now}
  </body>
</html>
```



output

LEARN
iGATE Learning, Education and Research Network

Step 3 : Configure a view resolver to tie the controller to the jsp:

JSP is usually used to render results. The ViewResolver maps the view name returned in the ModelAndView to a view. In our example, ViewResolver resolves "hello" to a jsp page. Spring provides different view resolvers that we shall see later. In our example, we have used the InternalResourceViewResolver.

The InternalResourceViewResolver prefixes the view name returned in the ModelAndView with the value of its prefix property (here /) and suffixes it with value from its suffix property – here .jsp. Since BasicSpringController returns a view name of "hello" in the ModelAndView , InternalResourceViewResolver will find the view at /hello.jsp.

4. Write the JSP that will render the homepage to the user.

We have added \${now} which is a EL (expression language) tag to the hello.jsp page. For people new to jstl, this is equivalent to inserting a scriptlet that will display current date and time (the variable being returned from ModelAndView object). Alternatively, you may also use the <c:out> jstl tag

Invoke the hello.jsp page as seen in the url address bar of the figure.

We will shortly see the working and execution, but first, let us discuss handler mappings.

4.3 :Spring MVC Components

Mapping Requests to Controllers

iGATE
ITOPS for Business Outcomes

➤ **DispatcherServlet consults a handler mapping bean which maps a specific controller to a URL pattern**

- BeanUrlHandlerMapping provides in-context resolving of controllers

```
<bean id="beanMapping"
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="order" value="1" />
</bean>
<bean name="/hello.obj" class="com.igate.ctrl.FirstController"/>
```

- SimpleUrlHandlerMapping : Its mappings property is wired with a java.util.Properties using <props>.

```
<bean id="firstCtrlr" class="com.igate.ctrl.FirstController"/>
<property name="order" value="2" />
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props> <prop key="/hello.obj">firstController</prop></props>
    </property>
</bean>
```

+ 9 +

LEARN
iGATE Learning, Education and Research Network

HandlerMappings: When associating a request with a specific controller, DispatcherServlet consults one or more handler mappings in order to know which controller to dispatch a request to. They typically map a specific controller to a URL pattern. The DispatcherServlet defaults to use BeanUrlHandlerMapping and DefaultAnnotationHandlerMapping (more on this later). See the first listing above for the xml code

Working with multiple handler mappings: Spring allows you to define multiple handler mappings. Which handler mapping gets used? All handler mapping classes implement Spring's Ordered interface. So, you can set the order properties of your handler mappings to indicate which has precedence in relation to others. In the case of multiple handler mappings, Spring will inspect each of them in order (with 1 having highest priority).

All of Spring MVC's handler mappings implement the **HandlerMapping** interface. Spring gives some ready-to-use implementations of this interface. The second listing above uses the SimpleUrlHandlerMapping, which maps controllers to URLs using a property collection defined in the context configuration file. It lets you map without having to name your beans in a special way.

SimpleUrlHandlerMapping property is wired with a java.util.Properties using <props>. The key element of each <prop> element is a URL pattern. Value is the name of the controller that will handle requests to the URL pattern.

4.3 :Spring MVC Components
ModelAndView Class

 iGATE
ITOPS for Business Outcomes

➤ This class fully encapsulates the view and model data that is to be displayed by the view. Eg:

ModelAndView("hello","now",now);

Map myModel = new HashMap();
myModel.put("now",now);
myModel.put("products",getProductManager().getProducts());
return new ModelAndView("product","model",myModel);

- Every controller returns a ModelAndView
- Views in Spring are addressed by a view name and are resolved by a view resolver

LEARN
iGATE Learning, Education and Research Network

The ModelAndView class allows you to specify a response to the client, resulting from actions performed in controller. This object holds both – the view the client will be presented with and the model used to render the view. Every controller must return a ModelAndView. See the two examples above.

The first parameter is the logical name of a view component that will be used to display the output from this controller. The next two parameters represent the model object that will be passed to the view and its value.

In the second example, the view component is product and the model is an object that is to be returned and its value is MyModel which is a HashMap object containing multiple values.

In the end, the dispatcher servlet needs a concrete view instance to render the view.

4.3 :Spring MVC Components

Resolving Views : The ViewResolver



The slide contains a table comparing four types of ViewResolvers:

View resolver	How it works
InternalResourceViewResolver	Resolves logical view names into View objects that are rendered using template file resources
BeanNameViewResolver	Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name
ResourceBundleViewResolver	Uses a resource bundle that maps logical view names to implementations of the View interface
XmlViewResolver	Resolves View beans from an XML file that is defined separately from the application context definition files

LEARN
iGATE Learning, Education and Research Network

So far, we have seen how model objects are passed to the view through the ModelAndView object. In Spring MVC, a view is a bean that renders results to the user. The view most likely is a JSP. But you could also use other view technologies like Velocity and FreeMarker templates or even views that produce PDF and MS-Excel documents.

View resolvers resolve the view name given by the ModelAndView object to a View bean. Spring provides a number of useful view resolvers, some of which are shown in the table above. See Spring docs for more.

InternalResourceViewResolver resolves a logical view name by affixing a prefix and a suffix to the view name returned by the ModelAndView object. It then loads a View object with the path of the resultant JSP. By default, the view object is an InternalResourceView, which simply dispatches the request to the JSP to perform the actual rendering. But, if the JSP uses JSTL tags, then you may replace InternalResourceView with JstlView as seen in the code demos earlier.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

Demo: DemoMVC_0



- This demo shows the cycle of a request in Spring MVC.
- Execute :
 - /hello.obj URL



- 12 -

Pls. refer to the **SpringWebDemo** project in demos. Invoke the first demo as :
<http://<server-name>:8080/<context-root>/hello.obj>

The /hello.obj url will be handled by DispatcherServlet which then looks for the basicspring-servlet.xml. In this configuration file, DispatcherServlet consults a handler mapping bean, the SimpleUrlHandlerMapping in this case. The BasicSpringController bean invokes its handleRequest() method, which returns a ModelAndView object. DispatcherServlet queries a ViewResolver to look up the View object that will render the response. The InternalResourceViewResolver resolves the view. Then DispatcherServlet dispatches the request to the View object hello.jsp, which is responsible for rendering a response back to the client.

Note: By changing the value `<property name="order" value="1" />` in `dispatcher-servlet.xml` file we can make out which HandlerMapper is being used. Refer the console logging.

4.4 : Spring MVC annotations

Implementing Controllers



```
@Controller
public class HelloController1 {
    @RequestMapping("/helloWorld")
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws .... {
        String now = new java.util.Date().toString();
        return new ModelAndView("hello", "now", now);
    }
}
```

```
@Controller
public class HelloController {
    @RequestMapping("/helloWorld")
    public String handleMyRequest(Map<String, Object> model) {
        String now = new java.util.Date().toString();
        model.put("now", now);
        return "hello";
    }
}
```

- 13 -



Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as `@Controller`, `@RequestMapping`, `@RequestParam`, `@ModelAttribute` etc. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. See the listing above for a simple example.

`@Controller` annotation indicates that this class is a controller class. This is a specialization of the `@Component` annotation. Thus `<context: component-scan>` will pick up and register `@Controller`-annotated classes as beans, just as if they were annotated with `@Component`.

`@RequestMapping` annotation serves two purposes. First, it identifies `handleRequest()` as a request-handling method. Second, it specifies that this method should handle requests whose path is `/helloWorld`. See the second listing above. Notice that `handleMyRequest()`, a user-defined method, replaces the `handleRequest()` method. `@RequestMapping` annotation identifies it as a request-handling method.

The `handleMyRequest()` takes a `Map` as a parameter, which represents the model—the data that's passed between the controller and a view. But the signature of a request-handling method can have anything as an argument. Notice that `handleMyRequest()` returns a `String` value which is the logical name of the view that should render the results. `ViewResolver` uses this to resolve actual view.

Note : `@RequestMapping` annotation can be used at class level too to map URLs onto an entire class. The `@RequestMapping` at class level defines the root URL path that the controller will handle. Method-level `@RequestMappings` narrow the scope of class-level `@RequestMapping`.

Demo

➤ Refer DemoMVC_1



HelloWorldController.java Insert title here X
http://localhost:8080/DemoMVC_1/hello.obj

Welcome to Spring!!!!

Time and Date : Thu May 23 10:16:15 IST 2013

- 14 -



Please refer to the DemoMVC_1 web project.

4.4 : Spring MVC annotations

Handling User Input



`@Controller
public class LoginFormController {
 @RequestMapping(value = "/login", method = RequestMethod.GET)
 public String onSubmit(@RequestParam("username") String username,
 @RequestParam("password") String password, Model model) {

 model.addAttribute("username", username);
 if (username.equals("majrul") && password.equals("majrul123"))
 return "success";
 else return "failure";
 } }`



Welcome to our Application!!
Login Valid! Welcome dear *majrul*



- 15 -

We have seen simple Controllers so far. But Controllers may need to perform business logic using information passed in as URL parameters or as submitted form data.

See the code listing above for an example. Here, `LoginFormController` is mapped to `/login` at the method level. This means that `onSubmit()` handles requests for `/login`. `"method"` attribute is set to `GET` indicating that this method will only handle HTTP GET requests for `/login`. The `onSubmit()` method takes two `String` parameters (`username` and `password`) and a `Model` object as parameters. The `username` parameter is annotated with `@RequestParam("username")` to indicate that it should be given the value of the `username` query parameter in the request. Same goes for `password` too. `onSubmit()` will use these parameters to authenticate user.

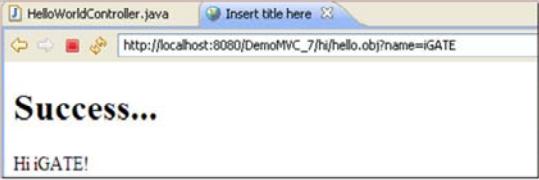
Now, see the `model` parameter. In earlier examples we passed in a `Map<String, Object>` to represent the model. Here we're using a new `Model` parameter. `Model` provides a few convenient methods for populating the model, such as `addAttribute()`. The `addAttribute()` method is similar to `Map`'s `put()` method, except that it finds out key part of the map by itself.

Notice how the values set in `addAttribute()` can be accessed in the jsp page:

```
<!-- success.jsp -->
<body>
<h1>Welcome to our Application!!</h1>
    Login Valid! Welcome dear <i><b>${username}</b></i>
</body>
```

Demo

➤ Refer DemoMVC_7 application



Success...

Hi iGATE!



LEARN
iGATE Learning, Education and Research Network

Please refer to the DemoMVC_7 web project.

Refer to HelloWorldController.java.

Invoke the application as :

http://localhost:8080/DemoMVC_7/hi/hello.obj?name=iGATE

4.4 : Spring MVC annotations

Processing forms : The controller class

```

@Controller
public class AddUserFormController {
    @RequestMapping(value = "/AddUser", method = RequestMethod.GET)
    public String showForm(Model model) {
        model.addAttribute(new User());
        return "addUser";
    }
    @RequestMapping(method = RequestMethod.POST)
    public String processForm(@Valid User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "failure";
        else {
            // some logic to persist user
            return "success";
        }
    }
}

```

addUser.jsp

iGATE
ITOPS for Business Outcomes

LEARN
iGATE Learning, Education and Research Network

Working with forms in a web application involves two operations: displaying the form and processing the form submission.

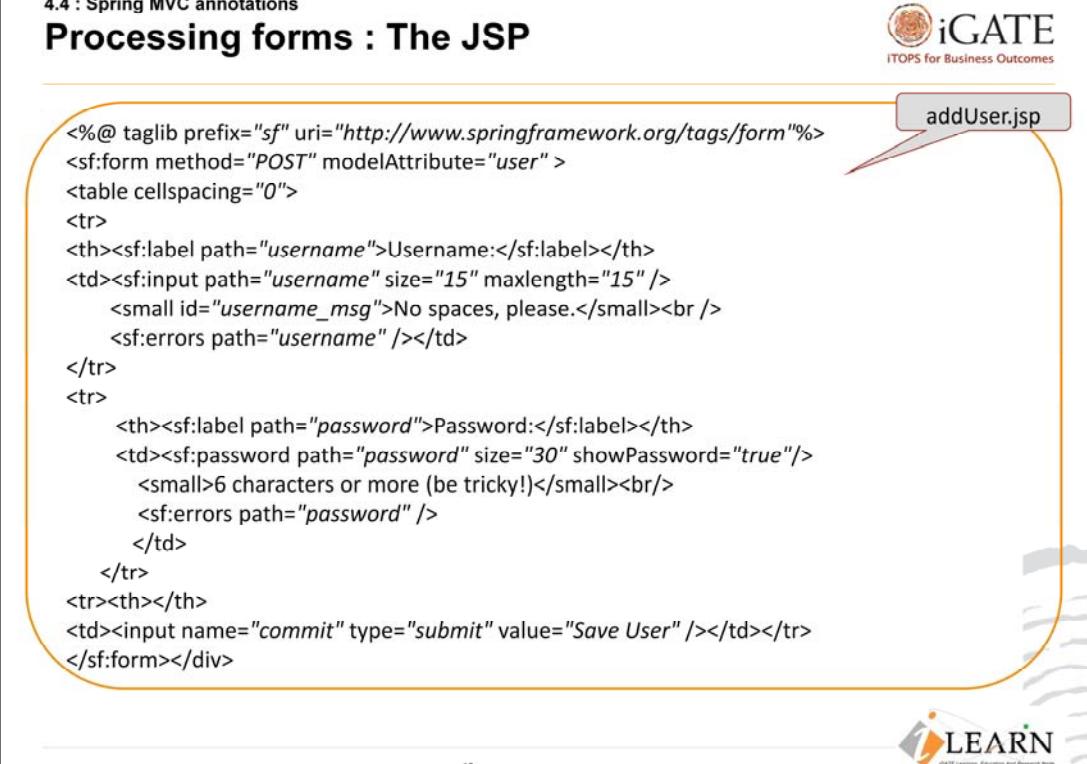
Our example allows to register new User. For this we have defined two handler methods to AddUserFormController to handle each of the operations. We first need to display the form in the browser before it can be submitted. The first

method (showForm()) displays the registration form. See the figure above.

Once the form is displayed, it'll need a User object to bind to the form fields. The showForm() method creates a User object and places it in the model.

4.4 : Spring MVC annotations

Processing forms : The JSP



```

<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="user" >
<table cellspacing="0">
<tr>
<th><sf:label path="username">Username:</sf:label></th>
<td><sf:input path="username" size="15" maxlength="15" />
    <small id="username_msg">No spaces, please.</small><br />
    <sf:errors path="username" /></td>
</tr>
<tr>
    <th><sf:label path="password">Password:</sf:label></th>
    <td><sf:password path="password" size="30" showPassword="true"/>
        <small>6 characters or more (be tricky!)</small><br/>
        <sf:errors path="password" />
    </td>
</tr>
<tr><th></th>
<td><input name="commit" type="submit" value="Save User" /></td></tr>
</sf:form></div>

```

LEARN
iGATE Learning, Education and Research Network

The addUser.jsp page uses Spring's form binding library. The `<sf:form>` tag binds the User object (identified by the `modelAttribute` attribute - see above) that `showForm()` placed into the model to the various fields in the form. The `<sf:input>` and `<sf:password>` tags have a `path` attribute that references the property of the User object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a User object and submitted to the server for processing.

Note that the `<sf:form>` specifies that it'll be submitted as an HTTP POST request. We thus now need another handler method that accepts POST requests. See the `processForm()` method in the code listing in the previous page which will process form submissions.

When the addUser.jsp form is submitted, the fields in the request will be bound to the User object (passed as an argument to `processForm()`). From there, some logic can be employed to persist user object into database.

Notice that User parameter is annotated with `@Valid`. This indicates that the User should pass validation before being persisted. We shall cover validation next.

4.4 : Spring MVC annotations

Validating input : declaring validation rules



```

public class User {
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20 characters long.")
    @Pattern(regexp = "^[a-zA-Z0-9]+$", message = "Username must be alphanumeric with no spaces")
    private String username;

    @Size(min = 6, max = 20, message = "The password must be at least 6 characters long.")
    private String password;

    @Pattern(regexp = "[A-Za-z0-9]+@[A-Za-z0-9.-]+[.][A-Za-z]{2,4}", message = "Invalid email address")
    private String email;

    //getter and setter methods for all these properties
}

```



- 19 -

Before an object can be processed further, it is essential to ensure that all the data in the object is valid and complete. Faulty form input must be rejected. For example, username must not contain spaces, password must be minimum 6 characters long, email must be correct etc.

The `@Valid` annotation (part of the JavaBean validation specification) tells Spring that the `User` object should be validated as it's bound to the form input. If anything goes wrong while validating the `User` object, the validation error will be carried to the `processForm()` method via the `BindingResult` that's passed in on the second parameter. If the `BindingResult`'s `hasErrors()` method returns true, then that means that validation failed.

How do we declare validation rules?

JSR-303 defines some annotations that can be placed on properties to specify validation rules. The code above shows the properties of the `User` class that are annotated with validation annotations.

`@Size` annotation validates that the fields meet criteria on their length.

`@Pattern` annotation along with a regular expression ensures that the value given to the `email` property fits the format of an email address and that the `username` is only made up of alphanumeric characters with no spaces.

Notice how we've set the `message` attribute with the message to be displayed in the form when validation fails. With these annotations, when a user submits a registration form to `AddUserFormController`'s `processForm()` method, the values in the `User` object's fields will be validated. If any of those rules are violated, then the handler method will send the user back to the form.

4.4 : Spring MVC annotations

Displaying validation errors

The diagram illustrates the Spring MVC validation process. It shows two code snippets: JSP code and Controller code. A callout labeled 'jsp' points to the JSP code, which includes a form field and an tag. Another callout labeled 'controller' points to the Controller code, which contains a method that checks for errors and returns a failure message if errors exist.

```
<td>
    <sf:password path="password" size="30" showPassword="true"/>
    <small>6 characters or more (be tricky!)</small><br/>
    <sf:errors path="password" />
</td>
```

```
public String processForm(@Valid User user, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "failure";
    }
    ....
```

We now need to tell jsp to display the validation messages.

We have passed the `BindingResult` parameter to `processForm()`. This knows whether the form had any validation errors via its `hasErrors()` method. But the actual error messages are also in there, associated with the fields that failed validation. To display those errors to the users, use Spring's form binding JSP tag library to display the errors. ie, the `<sf:errors>` tag can render field validation errors. See code listing above to see how this is done.

The `<sf:errors>` tag's `path` attribute specifies the form field for which errors should be displayed. For example, the above listing displays errors (if any) for the field whose name is `password`. If there are multiple errors for a single field, they'll all be displayed, separated by an HTML `
` tag.

Demo



➤ Refer the following Demos:

- DemoMVC_3
- DemoMVC_4
- DemoMVC_5
- DemoMVC_6
- DemoMVC_Complete



- 21 -

Lab



➤ Lab-3 from the lab guide



- 22 -

Summary



➤ We have so far seen:

- How to use Spring MVC architecture to build flexible and powerful web applications.
- Components like handler mappings, ViewResolvers and controllers
- MVC Annotations like @Controller, @RequestMapping and @RequestParam



- 23 -

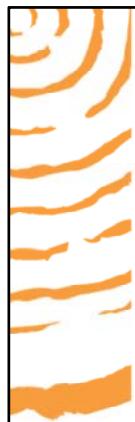
Review Questions



- **Question 1: If multiple handler mappings have been declared in an application, select the property that indicates which handler mapping has precedence?**
- Option 1: Order
 - Option 2: Sequence
 - Option 3: Index
 - Option 4: An application can't have multiple handler mappings
- **Question 2: To figure out which controller should handle the request, DispatcherServlet queries _____**
- Option 1: HandlerMappings
 - Option 2: ModelAndView
 - Option 3: ViewResolver
 - Option 4: HomeController

- 24 -





Spring Framework 3.0

Lesson 5: Aspect Oriented Programming (AOP)

Lesson Objectives



➤ Introduction to Spring AOP

- Learn AOP basics and terminologies
- Understand key AOP terminologies
- Understand the different ways that Spring supports AOP



- 2 -

Topics to be covered:

- Introduction to AOP
- AOP concepts
- AOP support in Spring using @AspectJ support
- AOP support in Spring using Schema-based AOP support

**5.1: Aspect-Oriented Programming (AOP)
Introduction to AOP**

iGATE
ITOPS for Business Outcomes

- AOP complements OOP
- Aspects enable the modularization of concerns that cut across multiple types and objects
- AOP complements Spring IoC to provide a very capable middleware solution

Implementation Modules

Persistence

Security

Logging

Implementation Modules

LEARN

- 3 -

Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. OOP decomposes applications into a hierarchy of objects; AOP decomposes programs into aspects and concerns. This allows modularization of concerns such as transaction management that would otherwise cut across multiple objects. Such concerns are often called cross-cutting concerns. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.).

There are two distinct types of AOP: static and dynamic. In static AOP, like AspectJ's AOP (<http://eclipse.org/aspectj/>), the crosscutting logic is applied to your code at compile time and you cannot change it without modifying the code and recompiling. With dynamic AOP like Spring's AOP, crosscutting logic is applied dynamically at run time. This allows you to make changes in the distribution of cross-cutting without recompiling the application.

The *AOP framework* is one of the key components of Spring. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

5.1: Aspect-Oriented Programming (AOP)
Understanding AOP: Example

 iGATE
ITOPS for Business Outcomes

➤ **An Example:**

```
void transfer(Account src, Account tgt, int amount) {  
    if (src.getBalance() < amount) {  
        throw new InsufficientFundsException();  
    }  
    src.withdraw(amount);  
    tgt.deposit(amount);  
}
```



AOP ensures that enterprise service code does not pollute application objects. For example, consider a banking application with a conceptually simple method for transferring an amount from one account to another as seen above. The transfer method simply checks if sufficient funds are available in the source account and if 'yes', performs the transfer. However, in a real-world banking application, this transfer method seems far from adequate. We must include security checks to verify that the current user has the authorization to perform this operation. We must enclose the operation in a database transaction to prevent accidental data loss. We must log the operation to the system log, and so on.

5.1: Aspect-Oriented Programming (AOP)

Understanding AOP: Example



void transfer(Account src, Account tgt, int amount) {
 if (!getCurrentUser().canPerform(OP_TRANSFER))
 throw new SecurityException();
 if (amount < 0)
 throw new NegativeTransferException();
 if (src.getBalance() < amount) {
 throw new InsufficientFundsException(); }
 Transaction tx = database.newTransaction();
 try {
 src.withdraw(amount);
 tgt.deposit(amount);
 tx.commit();
 systemLog.logOperation(OP_TRANSFER, src, tgt, amount);
 }
 catch(Exception e) { tx.rollback(); }



- 5 -

A very simplified version with all those new concerns would look somewhat like the code seen above.

We have now included a security check to verify that the current user has the authorization to perform this operation. We have also enclosed the operation in a database transaction in order to prevent accidental data loss.

The code has lost its elegance and simplicity because the various new concerns have become tangled with the basic functionality. The transactions, security, logging, etc. all exemplify cross-cutting concerns.

Therefore, we find that unlike the core concerns of the system, the cross-cutting concerns do not get properly encapsulated in their own modules. This increases the system complexity and makes maintenance considerably more difficult.

AOP attempts to solve this problem by allowing the programmer to develop cross-cutting concerns as full stand-alone modules called aspects.

What is Aspect ? It is modularization of a concern. For example, if I have a method to which I want to apply transaction, then applying transaction is a concern. If I modularize then I can use this concern (aspect) to many beans and methods. These services can then be applied declaratively to the components. Aspect-Oriented Programming has at its core the enabling of a better separation of concerns, by allowing the programmer to create cross-cutting concerns as first-class program modules.

5.1: Aspect-Oriented Programming (ASP)

AOP and Spring

➤ **AOP attempts to separate concerns, that is, break down a program into distinct parts that overlap in functionality sparingly.**

➤ **In particular, AOP focuses on the modularization and encapsulation of cross-cutting concerns.**

- 6 -

LEARN
iGATE Learning, Education and Research Network

Aspect Oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often, however, these components also carry additional responsibility beyond their core functionality. System services like logging, transaction management and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.

AOP makes it possible to modularize services and then apply them **declaratively** to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved.

The above figure represents a typical application that is broken down into modules. Each module's main concern is to provide services for its particular domain. However, each of these modules also requires similar support functionalities such as security, logging and transaction management. AOP presents an alternative that can be cleaner in many circumstances. With AOP, you can still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you are applying the new feature. Cross-cutting concerns can now be modularized into special objects called **aspects**.

5.1: Aspect-Oriented Programming (ASP)

AOP Terminology

iGATE
ITOPS for Business Outcomes

- **Aspect :**
 - the cross-cutting functionality being implemented
- **Advice :**
 - the actual implementation of aspect that is advising your application of a new behavior. It is inserted into application at joinpoints
- **Join-point :**
 - a point in the execution of the application where an aspect can be plugged in
- **Point-cut :**
 - defines at what joinpoints an advice should be applied
- **Target :**
 - the class being advised
- **Proxy :**
 - the object created after applying advise to the target

LEARN
iGATE Learning, Education and Research Network

AOP Concepts:

- **Aspect :** an aspect is the cross-cutting functionality you are implementing, that is, the area of your application you are modularizing. A simple example is logging. You can create a logging aspect and apply it throughout your application using AOP.
An aspect is also a combination of advice and pointcuts. This combination results in a definition of the logic that should be included in the application and where it should execute.
- **Joinpoint:** this is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown or even a field being modified. These are points where your aspect's code can be inserted into the normal flow of your application to add new behavior.
- **Advice:** this is the actual implementation of aspect, that is, the code that is executed at a particular joinpoint. It is advising your application of new behavior. There are many types of advices that we shall be seeing later.

5.1: Aspect-Oriented Programming (AOP)
AOP Terminology

iGATE
ITOPS for Business Outcomes

LEARN
iGATE Learning, Education and Research Network

- 8 -

- **Point-cut:** defines at what joinpoints an advice should be applied. Advice can be applied at any joinpoint supported by the AOP framework. By creating pointcuts, you gain fine-grained control over how you apply advice to the components in your application. A typical joinpoint is a method invocation. A typical pointcut is the collection of all method invocations in a particular class. You may not want to apply all aspects at all join-points. Point-cuts allow you to specify where you want advice to be applied.
- **Target:** is the class being advised. Without AOP, this class would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, this class is free to focus on its primary concern, oblivious to any advise being applied.
- **Proxy :** is the object created after applying advise to the target. As far as client objects are concerned, the target object (pre-AOP) and proxy object (post-AOP) are the same.
- **Weaving :** is the process of actually inserting aspects into the application code at the appropriate point.

In the above figure, the advice contains the cross-cutting behavior that needs to be applied. The join-points are well-defined points within the execution flow of the application that are candidates to have advice applied. The point-cut defines at what join-points that advice is applied.

5.1: Aspect-Oriented Programming (ASP)

AOP Terminology

iGATE
ITOPS for Business Outcomes

➤ **Types of advices:**

- Before advice
- After advice
- After-returning advice
- Around advice
- After-throwing advice

```

graph LR
    Client((Client calls abc())) --> Proxy((Proxy object))
    Proxy --> BEAN((BEAN abc(){...}))
    BEAN --> AfterCall((After method call))
    
```

LEARN
iGATE Learning, Education and Research Network

More on Advice: In the figure above, the client calls a method `abc()` on target object. But proxy object intercepts the call. The core goal of a proxy is to intercept method invocations and where necessary, execute chains of advice that apply to a particular method.

Spring's join-point model is built around method interception. This means that the Spring advice will be woven into the application at different points around a method's invocation. Since there are several points during the execution of a method, there are several advice types:

- **Before**—The advice functionality takes place before the advised method is invoked.
- **After**—The advice functionality takes place after the advised method completes, regardless of the outcome.
- **After-returning**—The advice functionality takes place after the advised method successfully completes.
- **After-throwing**—The advice functionality takes place after the advised method throws an exception.
- **Around**—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

5.2: AOP Support in Spring

AOP Frameworks

iGATE
ITOPS for Business Outcomes

- There are three dominant AOP frameworks:
 - AspectJ (<http://eclipse.org/aspectj>)
 - JBoss AOP (<http://www.jboss.org/jbossaop>)
 - Spring AOP (<http://www.springframework.org>)
- AOP support in Spring borrows a lot from the AspectJ project.
- Spring supports AOP in the following four flavors:
 - Classic Spring proxy-based AOP
 - @AspectJ annotation-driven aspects
 - Pure-POJO aspects
 - Injected AspectJ aspects (available in all versions of Spring)

variations on Spring's proxy-based AOP

LEARN
iGATE Learning, Education and Research Network

AOP frameworks differ from one another. Advice can be applied at the field modification level in some frameworks, whereas others only expose the join points related to method invocations. They may also differ in how and when they weave the aspects. In any case, ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework. The AOP frameworks have undergone some housecleaning in the last few years, resulting in some frameworks merging and others going extinct. Today, we have three dominant AOP frameworks (listed above).

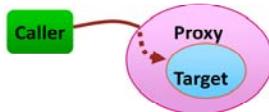
We shall focus on Spring AOP, this being a Spring course. But, the AOP support in Spring borrows a lot from the AspectJ project. Thus, Spring's support for AOP comes in four flavors (listed above). The first three are limited to method invocation. If you want more, consider implementing aspects in AspectJ. You can take advantage of Spring DI to inject Spring beans into AspectJ aspects.

Today, Spring's Classic Spring proxy-based AOP is no longer very popular, since Spring supports much cleaner and easier ways to work with aspects. Compared to simple declarative AOP and annotation-based AOP, Spring's classic AOP seems bulky and complex. Working directly with ProxyFactory- Bean can be wearying. Hence we will not cover this method in this course. However, you may refer to Appendix-D to understand the classic AOP API's better.

5.2: AOP Support in Spring
AOP Frameworks

iGATE
ITOPS for Business Outcomes

- **Key points of Spring's AOP framework:**
 - All advices are written in Java
 - Spring advises objects at runtime
 - Spring's AOP support is limited to method interception



LEARN
iGATE Learning, Education and Research Network

- All advices are written in Java
- Spring advises objects at runtime: Aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. See the figure above. The proxy class wraps around the target bean, intercepting advised method calls and forwarding those calls to the target bean. Spring doesn't create a proxied object until that proxied bean is needed by the application.
- Spring's AOP support is limited to method interception : Some other AOP frameworks, such as AspectJ and JBoss, provide field and constructor join points in addition to method pointcuts. Since Spring does not support field pointcuts, you cannot create very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there is no way to apply advice when a bean is instantiated. Method interception is suitable for most needs.

5.3: Bringing in @AspectJ

AspectJ's pointcut expression language

 iGATE
ITOPS for Business Outcomes

- AspectJ pointcut designators supported in Spring AOP

AspectJ	Description
args()	Limits matching to the execution of methods whose arguments are instances of the given types
@args()	Limits matching to the execution of methods whose arguments are annotated with the given annotation types
execution()	Matches join points that are method executions
this()	Limits matching to those where the bean reference of the AOP proxy is of a given type
target()	Limits matching to those where the target object is of a given type
@target	Limits matching to join points where the class of the executing object has an annotation of the given type
within()	Limits matching to join points within certain types
@within	Limits matching to join points within types that have the given annotation
@annotation	Limits matching to those where the subject of the join point has the given annotation



- 12 -

Pointcuts are used to pinpoint where an aspect's advice should be applied.

Pointcuts are one of the most fundamental elements of an aspect. In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language.

Note : *Spring only supports a subset of the pointcut designators available in AspectJ.*

The table above lists out the AspectJ pointcut designators that are supported in Spring AOP. All of these limit **Join point** matches to a particular scenario. Only the **execution** designator actually performs matches. The other designators are used to limit those matches.

5.3: Bringing in @AspectJ
AspectJ's pointcut expression language

• Writing pointcuts

`@Pointcut("execution(* training.spring.aop.*.(..))")`

Annotations pointing to parts of the pointcut:

- Returns any type
- any method
- Triggering method's execution
- type that method belongs to
- takes any arguments

`@Pointcut("execution(* training.spring.aop.*.(..))" && within(training.spring.aop))`

 iGATE
ITOPS for Business Outcomes

 LEARN
iGATE Learning, Education and Research Network

- 13 -

Writing Pointcuts: The pointcut expression shown above is used to apply advice whenever any method of any class in the training.spring.aop package is executed. The method specification starts with an asterisk, which indicates that we do not care what type the method returns. We need to specify the fully qualified class name and the name of the method we want to select. In the method's parameter list, the double-dot(..) indicates that the pointcut should select any method, no matter what the argument list is.
To confine the reach of that pointcut to only the training.spring.aop package, use the within() designator as seen in example above.

5.3: Bringing in @Aspect
Spring's @AspectJ support



package training.spring.aop;;
public interface Business {
 void doSomeOperation();
}

package training.spring.aop;
public class BusinessImpl implements Business {
 public void doSomeOperation() {
 System.out.println("I do what I do best, i.e sleep.");
 try { Thread.sleep(2000); }
 catch (InterruptedException e) {
 System.out.println("How dare you to wake me up?"); }
 System.out.println("Done with sleeping.");
 }
}



- 14 -

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. When the @AspectJ support is enabled, any bean in your container that has the @Aspect annotation will be automatically detected by Spring and used to configure Spring AOP.

To use @AspectJ aspects in a Spring configuration you need to enable Spring support, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will **automatically generate a proxy** for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support is enabled by including the following element inside your spring configuration:

<aop:aspectj-autoproxy/>

For this, you will need to bring in the necessary schema support. This is seen in the XML that is part of the subsequent demo.

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut and advice.

The code above shows the BusinessImpl class that contains business logic. We shall add Spring AOP to profile our business methods.

5.3: Bringing in @Aspect
Spring's @AspectJ support

The diagram illustrates the annotations used in the provided Java code. A red box labeled 'is the aspect' points to the '@Aspect' annotation. Another red box labeled 'defines a reusable pointcut within an aspect.' points to the '@Pointcut' annotation. The code itself is as follows:

```
@Aspect
public class BusinessProfiler {
    @Pointcut("execution(* training.spring.aop.*.*(..))")
    public void businessMethods() { }

    @Around("businessMethods()")
    public Object profile(ProceedingJoinPoint joinpoint) throws Throwable {
        long start = System.currentTimeMillis();
        System.out.println("Going to call the method.");
        Object output = joinpoint.proceed();
        System.out.println("Method execution completed.");
        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Method executed");
        return output;
    }
}
```

- 15 -

LEARN
iGATE Learning, Education and Research Network

See the code above.

Using **@AspectJ** annotation, we have declared that this class is an Aspect. This will profile our business method

Using **@Pointcut** annotation, we have defined a pointcut that will match the execution of all public method inside `training.spring.aop` package.

Using **@Around** annotation, we have defined a **Around** advice which will be invoked *before* and *after* our business method. The `pjp.proceed()` method calls our business method from **@Around** advice.

**5.3: Bringing in @Aspect
Spring's @AspectJ support**

 iGATE
ITOPS for Business Outcomes

```
<beans ...  
    xmlns:aop="http://www.springframework.org/schema/aop"  
    ...  
    http://www.springframework.org/schema/aop  
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">  
        <aop:aspectj-autoproxy /> <!-- Enable the @AspectJ support -->  
        <bean id="businessProfiler" class="training.spring.aop.BusinessProfiler" />  
        <bean id="myBusinessClass" class="training.spring.aop.BusinessImpl" />  
    </beans>
```

```
public class BusinessDemo {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("business.xml");  
        Business bc = (Business) context.getBean("myBusinessClass");  
        bc.doSomeOperation();  
    }  
}
```



In the Spring configuration file above, we have:

- Added the necessary AOP schemas.
- Enabled the @AspectJ support to our application using `<aop:aspectj-autoproxy />`
- Defined two normal Spring beans – one for our Business class and the other for Business Profiler (i.e. our aspect).

`<aop:aspectj-autoproxy/>` will **automatically generate proxy beans** whose methods match the pointcuts defined with `@Pointcut` annotations in `@Aspect`-annotated beans. To use the `<aop:aspectj-autoproxy>` configuration element, you'll need to remember to include the `aop` namespace in your Spring configuration file.

5.3: Bringing in @Aspect
Demo: DemoSpring_AOP2

 iGATE
ITOPS for Business Outcomes

➤ This demo shows how to apply cross-cutting functionality into Spring application using AOP with @AspectJ support



output

Going to call the method.
I do what I do best, i.e sleep.
Done with sleeping.
Method execution completed.
Method execution time...

From BusinessProfiler
From business logic
From BusinessProfiler

- 17 -



Please refer to demo,
Package -> com.igate.aop
Execute the BusinessDemo.java class

5.4: Schema-based AOP support

Declaring aspects in XML

 iGATE
ITOPS for Business Outcomes

AOP config element	Purpose
<aop:advisor>	Defines an AOP advisor.
<aop:after>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<aop:after-returning>	Defines an AOP after-returning advice.
<aop:after-throwing>	Defines an AOP after-throwing advice.
<aop:around>	Defines an AOP around advice.
<aop:aspect>	Defines an aspect.
<aop:aspectj-autoproxy>	Enables annotation-driven aspects using @AspectJ.
<aop:before>	Defines an AOP before advice.
<aop:config>	The top-level AOP element. Most <aop:> elements must be contained within <aop:config>.
<aop:declare-parents>	Introduces additional interfaces to advised objects that are transparently implemented.
<aop:pointcut>	Defines a pointcut.



- 18 -

From version 2.0 onwards, Spring offers support for defining aspects using the new "aop" namespace tags. The pointcut expressions and advice kinds supported are similar to @AspectJ style.

To use the aop namespace tags, you need to import the spring-aop schema. This is seen in the XML that is part of the subsequent demo.

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (multiple <aop:config> element can exist). An <aop:config> element can contain pointcut, advisor, and aspect elements, in that **order!**

5.4: Schema-based AOP support
Declaring aspects in XML

 iGATE
ITOPS for Business Outcomes

```
public class MyAdvice {  
    public void beforeMethodCall() {  
        System.out.println("Before Method Call");  
    }  
    public void aroundMethodCall() {  
        System.out.println("Around Method Call");  
    }  
    public void afterMethodCall() {  
        System.out.println("After Method Call");  
    }  
    public void afterException() {  
        System.out.println("After Exception thrown");  
    }  
}
```

```
<bean id="advice" class="training.spring.schemaAOP.MyAdvice" />
```



Let us see an example of using Spring's aop configuration namespace. The code listing above is very straightforward and simply creates functions of an advice. It's a POJO with a handful of methods. Thus we can register it as a bean in the Spring application context like any other class. See above for how. The MyAdvice class seems an Aspect but needs Spring's AOP magic to become one.

5.4: Schema-based AOP support

Declaring aspects in XML



- 20 -



```
<beans ... >
<bean id="advice" class="training.spring.schemaAOP.MyAdvice" />
<bean id="myBusinessClass" class="training.spring.schemaAOP.BusinessImpl" />
<aop:config>
  <aop:aspect ref="advice">
    <aop:before
      pointcut="execution(* training....BusinessImpl.doBusiness(..))"
      method="beforeMethod" />
    <aop:around
      pointcut="execution(* training....BusinessImpl.doBusiness(..))"
      method="aroundMethod" />
    ...
    <aop:after-throwing
      pointcut="execution(* training....BusinessImpl.doBusiness(..))"
      method="afterException" />
  </aop:aspect>
</aop:config>
</beans>
```

Using the Spring's AOP configuration elements, as shown in the listing above, we have converted the MyAdvice POJO into an aspect!

Notice that most of the configuration elements are used within the context of the `<aop:config>` element. Within `<aop:config>` you may declare one or more advisors, aspects, or pointcuts. In our example, we have declared a single aspect using the `<aop:aspect>` element. The `ref` attribute references the POJO bean (MyAdvice in this case) that will be used to supply the functionality of the aspect ie methods called by any advice in the aspect. The aspect has several bits of advice. The `<aop:before>` element calls aspect method (`beforeMethod()`) before the methods matching the pointcut are executed. The `<aop:after-throwing>` element defines an *after-throwing advice* to call the `afterException()` if any exceptions are thrown.

In all advice elements, the `pointcut` attribute defines the pointcut where the advice will be applied. Notice that the value of the `pointcut` attribute is the same for all of the advice elements. That's because all of the advice is being applied to the same pointcut.

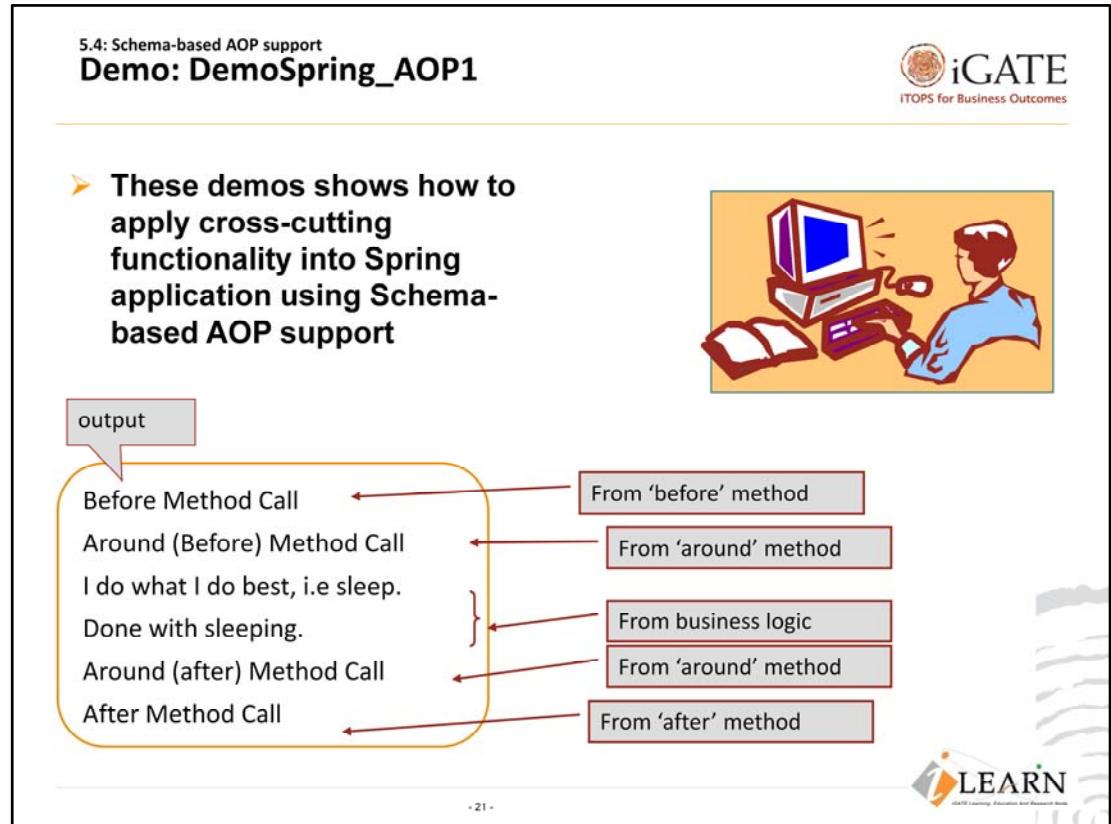
The pointcut expression is repeated throughout the code. To avoid this, you can use `<aop:pointcut>` element to define a named pointcut that can be used by all of the advice elements.

```
<aop:aspect ref="advice">
  <aop:pointcut id="pt" expression=
    "execution(* training.spring.schemaAOP.BusinessImpl.doBusiness())" />
  <aop:before pointcut-ref="pt" method="beforeMethod" />
```

5.4: Schema-based AOP support
Demo: DemoSpring_AOP1



➤ These demos shows how to apply cross-cutting functionality into Spring application using Schema-based AOP support



- 21 -



Please refer to demo, DemoSpring_AOP1 -> com.igate.aop
 Execute the BusinessDemo.java class

We have modified the aroundMethod() method in MyAdvice.java a bit to add the proceed() method. The ProceedingJoinPoint object allows us to invoke the advised method from within our advice. The advice method will do everything it needs to do and, when it's ready to pass control to the advised method, it'll call ProceedingJoinPoint's proceed() method. If you don't include the call to proceed() method, your advice will block access to the advised method.

5.5: Logging as an Aspect

Integrating Log4j framework into AOP



➤ **Logging has a lot of characteristics that make it a prime candidate for implementation as an aspect:**

- Logging code is often duplicated across an application, leading to a lot of redundant code across multiple components in the application.
- Logging logic does not provide any business functionality; it's not related to the domain of a business application.

- 22 -



Logging has a lot of characteristics that make it a prime candidate for implementation as an aspect. The following are two of the notable characteristics:

Logging code is often duplicated across an application, leading to a lot of redundant code across multiple components in the application. Even if the logging logic is abstracted to a separate module so that different components have a single method call, that single method call is duplicated in multiple places. Logging logic does not provide any business functionality; it's not related to the domain of a business application.

Spreading the logging across multiple components introduces some complexity to the code. Business objects not only perform business tasks, they also take care of logging functionality (and other such crosscutting concerns, such as security, transaction, caching, etc.).

AOP makes it possible to modularize these crosscutting services and then apply them declaratively to the components that they should affect.

Let us see a simple example. We shall use a around advice AOP advice to intercept a method in the business class and log the operation at DEBUG level.

5.5: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

iGATE
ITOPS for Business Outcomes

```
public interface SampleInterface {  
    public void process();  
    public String getName();  
    public int getAge();  
    public void setAge(int age);  
    public void setName(String str);  
}  
  
public class SampleBean implements SampleInterface {  
    private String name;  
    private int age;  
    //getter/setter methods for these properties  
  
    public void process() {  
        System.out.println("checking with the process() method-1");  
    }  
}
```

Business interface and its implementing class

Business method

LEARN
iGATE Learning, Education and Research Network

- 23 -

5.5: Logging as an Aspect

Eg : Integrating Log4j framework into AOP



```
public class LoggingInterceptor {  
    Logger myLog;  
    public Object logs(ProceedingJoinPoint call) throws Throwable {  
        Object point = null;  
        myLog = Logger.getLogger(LoggingInterceptor.class);  
        PropertyConfigurator.configure("log4j.properties");  
        try {  
            System.out.println("from logging aspect: entering method "  
                + call.getSignature().getName());  
            myLog.info("Hello : It is " + new java.util.Date().toString());  
            point = call.proceed();  
            System.out.println("from logging aspect: exiting method ");  
        } catch (Exception e) {  
            System.out.println("Logging the exception with date " + new Date());  
        }  
        return point;  
    }  
}
```

The interceptor

- 24 -



5.5: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

The configuration file

```
<beans .....>
    <bean id="sampleBean" class="training.spring.aop.logger.SampleBean"/>
    <bean id="loggingInterceptor"
          class="training.spring.aop.logger.LoggingInterceptor" />
    <aop:config>
        <aop:aspect ref="loggingInterceptor">
            <aop:pointcut id="myCutLogging" expression="execution(* *.p*(..))"/>
            <!-- - when you want to do? before method ,after method,..... -->
            <aop:around pointcut-ref="myCutLogging" method="logs" />
        </aop:aspect>
    </aop:config>
</beans>
```

log4j.properties

```
log4j.rootLogger=debug, myAppender
log4j.appender.myAppender=org.apache.log4j.ConsoleAppender
log4j.appender.myAppender.layout=org.apache.log4j.SimpleLayout
```

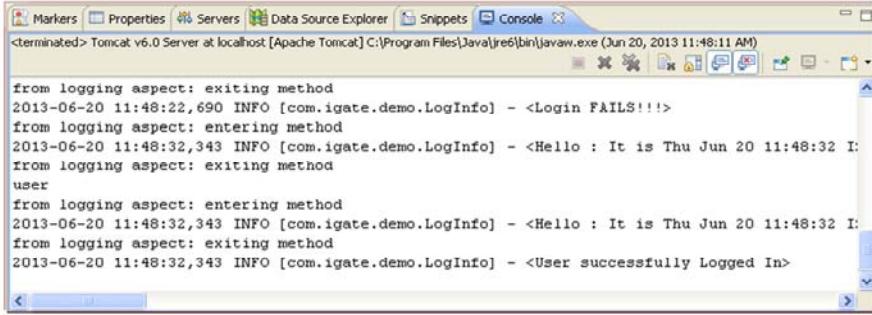
LEARN

- 25 -

Demo: DemoMVC_AOP

 iGATE
ITOPS for Business Outcomes

➤ This demo shows how to integrate the Log4j logging framework with AOP using MVC based an application




output

 LEARN
iGATE Learning, Education and Research Network

Please refer to demo,

Lab



➤ Lab-4 from the lab guide



- 27 -

Summary



➤ We have so far seen:

- AOP basics and terminologies
- Key AOP terminologies
- The different ways that Spring supports AOP.



- 28 -



AOP is a powerful complement to object-oriented programming. With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules. You can then declaratively or programmatically define exactly where and how this behavior is applied. This reduces code duplication and lets your classes focus on their main functionality. Thus, the AOP technique helps you add design and run-time behavior to an object model in a non-obtrusive manner by using static and dynamic crosscutting.

The first thing that comes to mind when someone mentions AOP is logging, followed by transaction management. However, these are just special applications of AOP. AOP can be used for performance monitoring, call auditing, caching and error recovery too. More advanced uses of AOP include compile-time checks of architecture standards. For example, you can write an aspect that will enforce you to call only certain methods from certain classes.

Today, Spring AOP offers fine grained object advising capabilities using AspectJ support.

Review Questions



➤ **Question 1: In Spring's aop configuration namespace, how is an aspect defined?**

- Option 1 : <aop:advisor>
- Option 2 : <aop:aspect>
- Option 3 : <aop:declare-aspect>
- Option 4 : <aop:config>



➤ **Question 2 : In addition to method join points, Spring also supports field and constructor joinpoints.**

- Option 1 : True
- Option 1 : False



- 29 -

Review Questions



➤ **Question 3 : ProceedingJoinPoint's _____ method must be used to provide access to the advised method, so that it can execute.**

- Option 1 : invoke()
- Option 2 : continue()
- Option 3 : proceed()
- Option 4 : next()



➤ **Question 4 : <aop:aspectj-autoproxy/> automatically proxies beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans.**

- Option 1 : True
- Option 2 : False

- 30 -



Review Questions





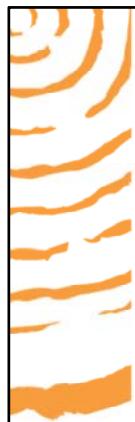
S	T	A	R	G	E	T	P
J	P	D	B	C	A	A	R
N	R	V	I	U	T	S	O
P	O	I	N	T	C	U	T
R	C	C	V	W	E	Y	E
O	E	E	O	Q	P	F	E
X	E	T	K	G	S	M	G
Y	D	E	E	G	A	H	E



- 31 -

There are some AOP terms hidden in the above table. The words may be across, down or even upside down.

1. the cross-cutting functionality being implemented
2. the actual implementation of aspect that's advising your application of new behavior
3. defines at what joinpoints, an advice should be applied
4. the class being advised
5. the object created after applying advise to the target
6. A method of the MethodInterceptor interface
7. Helps to go to the next interceptor in the chain



Spring Framework 3.0

Lesson 6: Testing Spring applications



Lesson Objectives



➤ Understand Spring's Test Context framework

- Understand unit testing in Spring framework
- Learn how Unit testing can be done at POJO level.



- 2 -

Testing is a crucial part of software development. One should never release software that has not been tested in some way. The easiest testing technique is unit testing. It allows you to test each component of the application separately, in a controlled environment. Once these tests are successful, you may perform integration testing. However, these are difficult to implement, especially if you are dealing with EJB's or JTA transactions. Let us see the value-add that the adoption of the IoC principle brings to unit testing and the benefits that the Spring Framework provides in integration testing.

One of the main benefits of Dependency Injection is that your code should really depend far less on the container than in traditional J2EE development. The POJOs that comprise your application should be testable in JUnit or TestNG tests, with objects simply instantiated using the new operator, *without Spring or any other container*. You can use mock objects (in conjunction with many other valuable testing techniques) to test your code in isolation. For example, you will be able to test service layer objects by stubbing or mocking DAO or Repository interfaces, without any need to access persistent data while running unit tests.

Introduction to testing in Spring

➤ **Spring Framework provides some mock objects for unit testing scenarios:**

- **JNDI** : useful for setting up a simple JNDI environment for test suites or stand-alone applications.
- **Servlet API** : contains Servlet API mock objects, targeted at usage with Spring's Web MVC framework - useful for testing web contexts and controllers.
- **Portlet API** : The portlet package contains Portlet API mock objects, targeted at usage with Spring's Portlet MVC framework.

➤ **Unit testing support classes**

- **General utilities** : ReflectionTestUtils class used in unit and integration testing scenarios
- **Spring MVC** : ModelAndViewAssert class used for unit tests dealing with Spring MVC ModelAndView objects.

True unit tests typically will run extremely quickly, as there is no runtime infrastructure to set up, whether application server, database, ORM tool, or whatever.

For certain unit testing scenarios, the Spring Framework provides the following mock objects and testing support classes.

Mock objects

JNDI : The org.springframework.mock.jndi package contains an implementation of the JNDI SPI, which is useful for setting up a simple JNDI environment for test suites or stand-alone applications. If, for example, JDBC DataSources get bound to the same JNDI names in test code as within a J2EE container, both application code and configuration can be reused in testing scenarios without modification.

Servlet API : The org.springframework.mock.web package contains a comprehensive set of Servlet API mock objects, targeted at usage with Spring's Web MVC framework, which are useful for testing web contexts and controllers. These mock objects are generally more convenient to use than dynamic mock objects (e.g., EasyMock) or existing Servlet API mock objects (e.g., MockObjects).

Portlet API : The org.springframework.mock.web.portlet package contains a set of Portlet API mock objects, targeted at usage with Spring's Portlet MVC framework.

Spring Framework also provides the following testing support classes for unit testing scenarios:

- General utilities :** The org.springframework.test.util package contains ReflectionTestUtils, which is a collection of reflection-based utility methods for use in unit and integration testing scenarios in which the developer would benefit from being able to set a non-public field or invoke a non-public setter method when testing application code involving, for example:
- ORM frameworks such as JPA and Hibernate which condone the usage of private or protected field access as opposed to public setter methods for properties in a domain entity
 - Spring's support for annotations such as @Autowired and @Resource which provides dependency injection for private or protected fields, setter methods, and configuration methods

Spring MVC : The org.springframework.test.web package contains ModelAndViewAssert, which you can use in combination with JUnit 4+, TestNG, etc for unit tests dealing with Spring MVC ModelAndView objects.

Unit testing Spring MVC Controllers: To test your Spring MVC Controllers, use ModelAndViewAssert combined with MockHttpServletRequest and MockHttpSession from the org.springframework.mock.web package.

JDBC testing support : The org.springframework.test.jdbc package contains SimpleJdbcTestUtils, which is a collection of JDBC related utility functions intended to simplify standard database testing scenarios.

6.2: Spring TestContext Framework
Framework basics



- Provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use.
- The core of the framework consists of:
 - **TestContextManager** class: manages a TestContext
 - **TestContext** class : responsible for holding the context of the current test
 - **TestExecutionListener** interface : instrument the actual test execution
- **TestExecutionListener implementations that are configured by default:**
 - **DependencyInjectionTestExecutionListener** : support dependency injection of the test instance
 - **DirtiesContextTestExecutionListener**: handles @DirtiesContext annotation
 - **TransactionalTestExecutionListener**. Handles transactional test execution with default rollback semantics.



- 5 -

The *Spring TestContext Framework* (`org.springframework.test.context`) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use. It provides many defaults that can be overridden through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit (all versions) and TestNG 5.10 in the form of abstract support classes. For JUnit 4.5+, the framework also provides a custom Runner that allows one to write test classes that are not required to extend a particular class hierarchy.

The core of the framework consists of the `TestContext` and `TestContextManager` classes and the `TestExecutionListener` interface.

For more details on this framework, please refer to the Spring reference documentation and the Javadoc that ships with Spring.

```
@DirtiesContext
public void testProcessWhichDirtiesAppCtx() {
    // some logic that results in the Spring container being dirtied }
```

6.2: Spring TestContext Framework

Spring Test Example

```

public class Writer {
    int number;
    // getter and setter methods for number
}

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("unittest.xml")
public class MyTest {
    @Autowired
    private Writer writer;

    @Test
    public void createTest() {
        System.out.println("in test class");
        Assert.assertNotNull(writer.getNumber());
    }
}

```

The XML configuration file (unittest.xml) contains:

```

<beans>
    <bean id="writer" class="Writer">
        <!-- configuration for Writer -->
    </bean>
</beans>

```

JUnit Results:

- Runs: 1/1
- Errors: 0
- Failures: 0

LEARN

The Spring TestContext Framework provides consistent loading of Spring ApplicationContexts and caching of those contexts. This support is important, because startup time can become an issue - because the objects instantiated by the Spring container take time to instantiate.

Each TestContext provides context management and caching support for the test instance for which it is responsible. Test instances do not automatically receive access to the configured ApplicationContext. However, configuration is achieved merely by declaring the `@ContextConfiguration` annotation at the class level. Let us see how.

JUnit4 defines `@RunWith` annotation which expects a class that implements `org.junit.runner.Runner` interface. Spring 2.5+ provides `SpringJUnit4ClassRunner` that implements this interface. Spring also provides the `@ContextConfiguration` annotation which points to the location of the application context configuration XML file.

The code snippet above shows how a simple test case can be written by wiring beans from the provided and declared xml.

By default, once loaded, the configured ApplicationContext is reused for each test. Thus the setup cost is incurred only once (per test fixture), and subsequent test execution is much faster. In case a test dirties (modifies) the application context, requiring reloading - eg, changing a bean definition or the state of an application object -- you can annotate your test method with `@DirtiesContext`. This causes the test fixture to reload the configurations and rebuild the application context before executing the next test.

6.2: Spring TestContext Framework

Dependency Injection of test fixtures



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("unittest.xml")
public class MyDAOTest {
    @Autowired
    private EmployeeJdbcDAO employeeDAO;
    @Test
    public void createTest() {
        Assert.assertNotNull(employeeDAO.getSalarySum());
    }
}
```

```
<bean id="myDataSource" class=".....DriverManagerDataSource">
    .....
</bean>
<bean id="employeeDAO" class="training.spring.EmployeeJdbcDAO">
    <property name="dataSource"><ref local="myDataSource"/></property>
</bean>
```

autowired



+ 7 +

When the TestContext framework loads your application context, it can optionally configure instances of your test classes via Dependency Injection. This provides a convenient mechanism for setting up test fixtures using preconfigured beans from your application context. A strong benefit here is that you can reuse application contexts across various testing scenarios (e.g., for configuring Spring-managed object graphs, transactional proxies, DataSources, etc.), thus avoiding the need to duplicate complex test fixture set up for individual test cases.

See the above listing where EmployeeJdbcDAO class performs data access logic for the Emp domain object.

Demo



➤ **Execute the following demos**

- MyTest.java
- MyDAOTest.java



= 8 =

Please refer the following demos MyTest.java and MyDAOTest.java

Lab



- **Unit test all your lab applications done so far.**



- 9 -

Add the notes here.

Lesson Summary



➤ We have so far seen that:

- Spring supports both – unit and integration testing
- Unit testing can be done at POJO level and at web-layer level.



~ 10 ~

Test driven development is a practical way of ensuring the correctness of the code you write. Testing frameworks like JUnit are valuable in writing and running automated test cases. Although Spring supports unit testing by encouraging decoupled code, Spring isn't often directly involved in testing activity. But Spring also provides the TestContext framework to ease testing.

Spring Framework ver 3.0

Copyright © 2011 iGATE Corporation. All rights reserved. No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of iGATE Corporation.

iGATE Corporation considers information included in this document to be Confidential and Proprietary.

Document Revision History

Date	Revision No.	Author	Summary of Changes
Aug-2012	2.0	Mohan C	Lab book exercises are revamped
June-2013	3.0	Mohan C	Lab book exercises are revamped

Table of Contents

<i>Document Revision History</i>	1
<i>Table of Contents</i>	2
<i>Getting Started</i>	3
<i>Overview</i>	3
<i>Setup Checklist for Spring Framework</i>	3
<i>Minimum System Requirements</i>	3
<i>Creating the first Spring application:</i>	3
<i>Lab 1. Injecting dependencies into a Spring application</i>	4
<i>Lab 2. Spring JDBC</i>	8
<i>Lab 3. Spring MVC</i>	9
<i>Lab 4. Injecting cross-cutting concerns</i>	16
<i>Lab 5. Review, Testing and Debugging</i>	17
<i>Appendices</i>	18
<i>Appendix A: Class Diagrams</i>	18

Getting Started

Overview

This lab book is a guided tour for learning Spring Framework version 3.0. It comprises solved examples and 'To Do' assignments. Follow the steps provided in the solved examples and work out the 'To Do' assignments given.

Setup Checklist for Spring Framework

Here is what is expected on your machine in order for the lab assignments to work.

Minimum System Requirements

- Intel Pentium IV or higher
- Microsoft Windows (NT 4.0/XP/2K)
- Memory: 256MB of RAM (512 recommended)
- 500MB hard disk space
- JDK version 1.5 + with help, Netscape or IE
- MS-Access/Connectivity to Oracle database
- JBoss AS 4.2 +
- Eclipse 3.5+
- Spring 3.0 API from <http://www.springsource.org/download>. Download spring-framework-3.0.0.RELEASE-with-docs.zip, which contains the documentation also and unzip it.

Creating the first Spring application:

- ✓ Ensure that Java 5 or 6 is installed and Eclipse 3.5 is available.
- ✓ You will need JBoss AS 4.2 + to work with.
- ✓ Unzip the spring-framework-3.0.4.RELEASE-with-docs.zip into any folder.
- ✓ Create a new project in JBoss. Name it.

Lab 1. Injecting dependencies into a Spring application

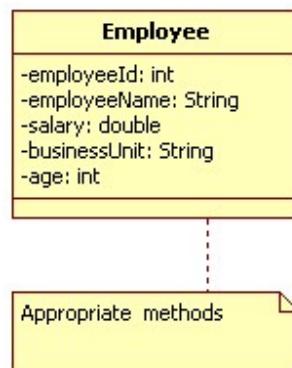
Goals	<ul style="list-style-type: none"> Using IoC to integrate disparate systems in a loosely coupled manner.
Time	180 minutes

Problem statement-1.1: Injecting dependencies

Write an Employee bean. Inject values into bean using DI and display all values. Refer the class diagram below

The output would look as shown below:

```
<terminated> EmpClient (2) [Java Application] C:\Program Files
Employee details
-----
Employee ID : 12345
Employee Name : Harriet
Employee Salary : 40000.0
Employee BU : PES-BU
Employee Age : 30
```



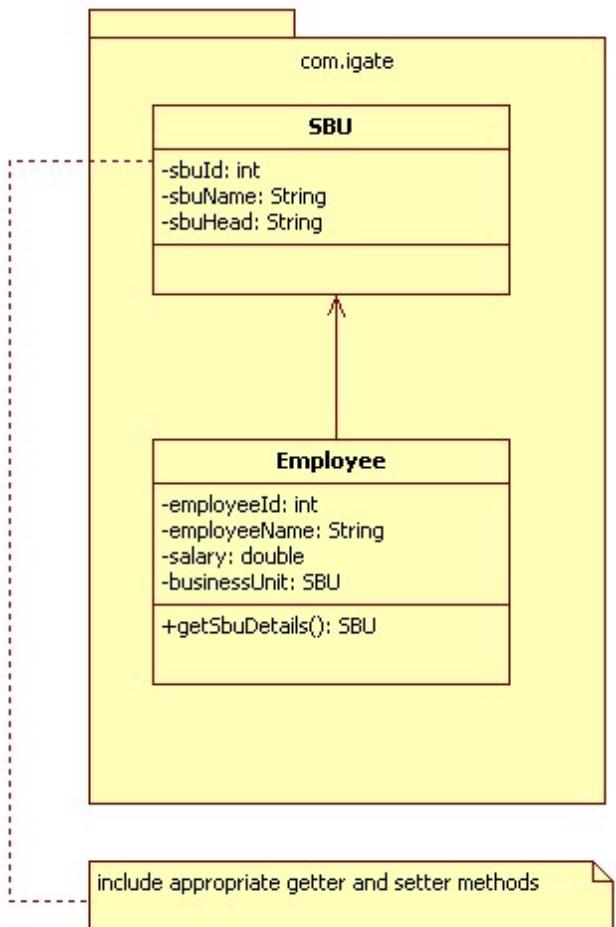
Class Diagram 1: Employee



Keep each of the lab solutions separate, preferably in different packages/source folders

Problem statement-1.2: Injecting dependencies

Code SBU bean. Revisit the Employee bean and provide a method to retrieve SBU details (getSBUDetails()) for the employee. You will need to inject the SBU bean to the Employee bean as shown in the Class diagram below:



Class Diagram 2: SBU and Employee

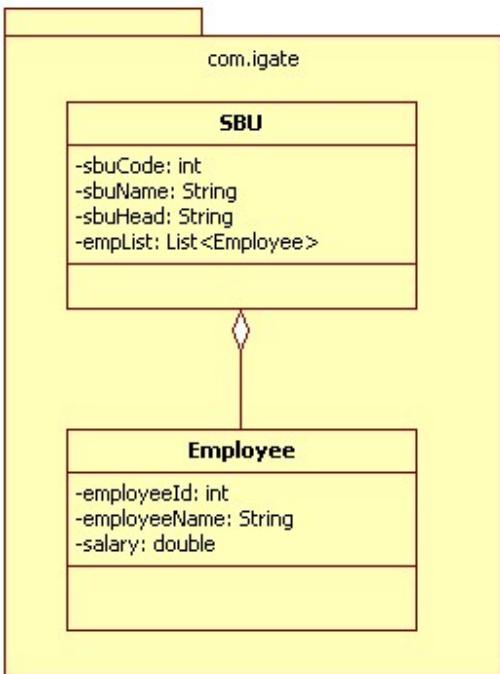
The output would look as shown below:

```

Problems @ Javadoc Declaration Console <terminated> EmpClient (3) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 20, 2012 4:30:57 PM)
Employee details
-----
Employee [empAge=40, empId=12345, empName=Harriet, empSalary=40000.0
sbu details=SBU [sbuCode=PES-BU, sbuHead=Kiran Rao, sbuName=Product Engineering Services]]
  
```

Problem statement-1.3: Injecting dependencies

Revisit the SBU bean. Create a new property called empList which will contain a list of all employees in the PES BU. Display the SBU details, followed by a list of all employees in that BU. To inject employee objects into the SBU bean, use “List” collection. Allocate two employees to PES. Refer Class diagram below



Class Diagram 3: SBU and Employee (Ver -2)

The output would look as shown below:

```

Problems @ Javadoc Declaration Console 
<terminated> EmpClient (4) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 20, 2012 4:53:43 PM)
SBU details
-----
sbuCode=PES-BU, sbuHead=Kiran Rao, sbuName=Product Engineering Services
Employee details:-----
[Employee [empAge=30, empId=12345, empName=Harriet, empSalary=40000.0], Employee [empAge=30, empId=12346

```

Problem statement-1.4: Injecting dependencies

Develop a console based spring application where main method of client class will retrieve employee information from Employee table and displays info in the console as shown:

Input:

Employee ID : 100

Output:

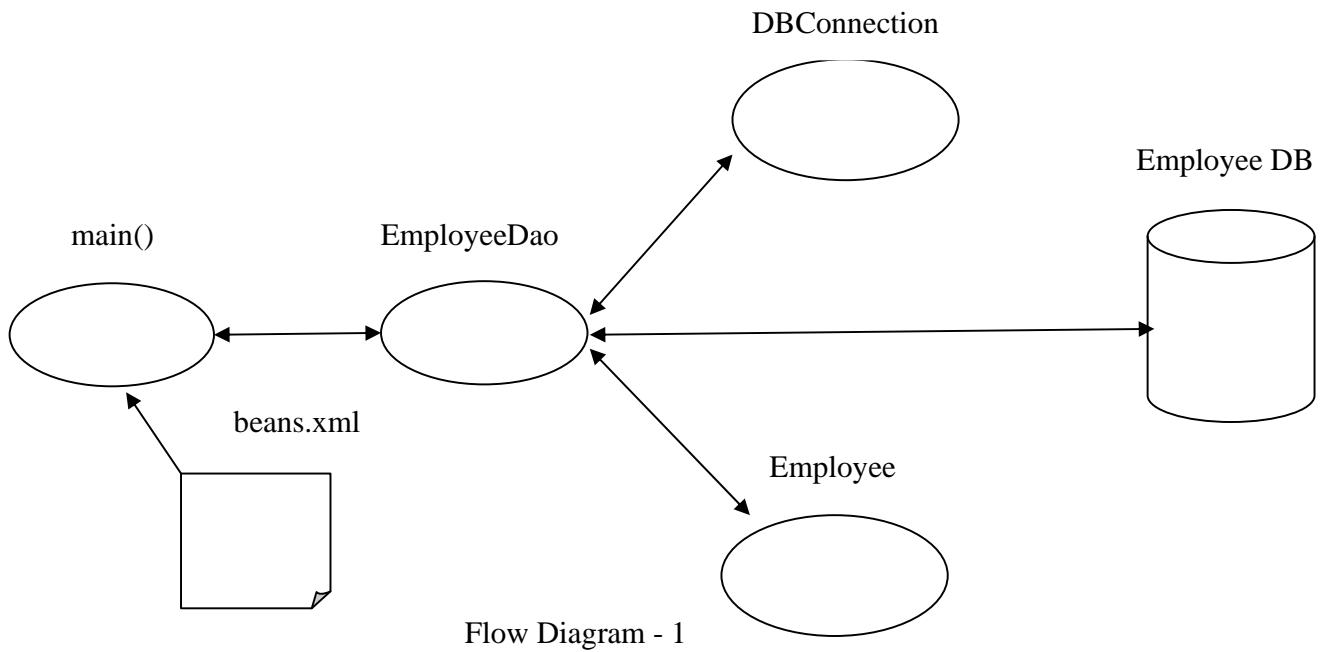
Employee Info:

Employee ID :100

Emplployee NAME :Rama

Employee SALARY :12345.67

Refer diagram below for implementation details:



Note: implement above application using

- Setter Injection
- Constructor Injection (use index and type attribute with constructor arg tag)
- Use different bean wiring mechanism like..
 - By Name
 - By Type
 - Auto Wiring

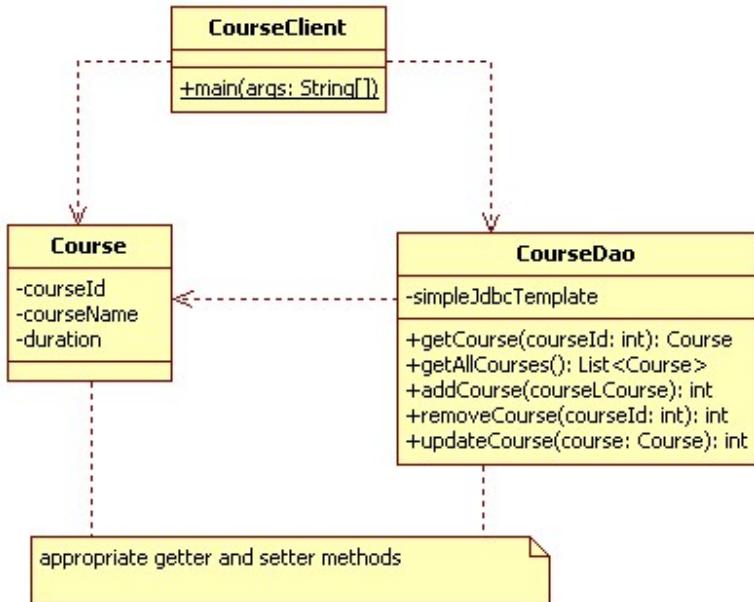
Change Request: Now Change the application so that all components are autowired and components are automatically scanned.

Lab 2. Spring JDBC

Goals	<ul style="list-style-type: none"> Demonstrate Spring's JDBC and DAO support
Time	180 minutes

Problem Statement-1: Using Spring SimpleJdbcTemplate

Develop a Spring JDBC Template based application to manage list of Courses by a Trainer. Refer the class diagram below to develop required classes



Class Diagram 4: Course Related Classes

Problem statement-2: Using Spring JdbcDaoSupport class

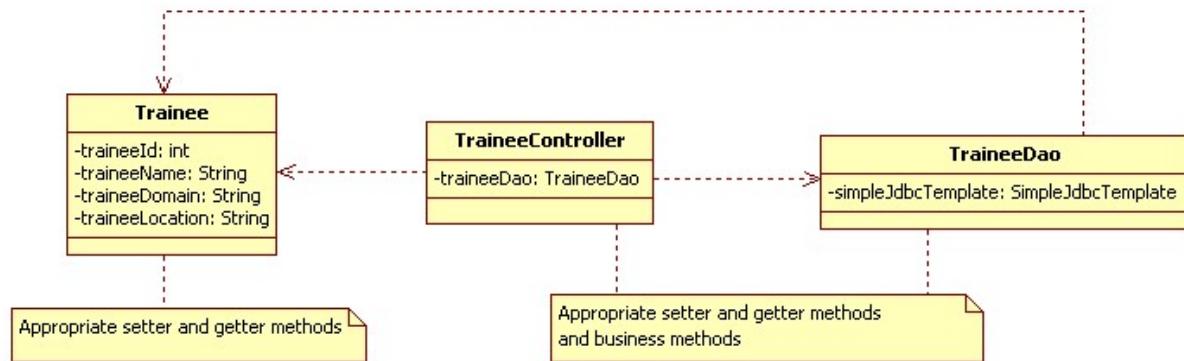
Redo the above problem using `JdbcDaoSupport` class.

Lab 3. Spring MVC

Goals	<ul style="list-style-type: none"> Demonstrate Spring's MVC framework
Time	120 minutes

Version -1:

Develop a Spring MVC based application to manage list of trainees by an admin. Refer the class diagram below to develop required classes.



Class Diagram 5: Trainee related Classes

Initially when the application is deployed login page should come up as shown below.

The login page interface consists of a title 'Login Page' at the top, followed by two input fields labeled 'Username' and 'Password' with their respective text boxes, and a 'Login' button at the bottom.

Administrator will enter valid credentials to go to the page shown below.

Trainee Management System

Pick your operation
Add a Trainee
Delete a Trainee
Modify a Trainee
Retrieve a Trainee
Retrieve all Trainees

Admin select add hyperlink to add a new trainee. The following page shows up.

Enter trainee details

Trainee ID	<input type="text"/>
Trainee Name	<input type="text"/>
Trainee Location	<input type="text"/>
Trainee Domain	<input type="text"/>
<input type="button" value="addTrainee"/>	

After entering trainee details admin will submit, to insert trainee info into database.

If admin selects delete operation refer the following screen shots to design your application.

Delete Operation

Please enter trainee ID	<input type="text"/>	<input type="button" value="delete"/>
-------------------------	----------------------	---------------------------------------

After entering trainee ID in the same page trainee info will be displayed as shown below

Delete Operation

Please enter trainee ID

Trainee Info

Trainee ID	Trainee Name	Trainee Location	Trainee Domain
100	Rama	Mumbai	JEE
<input type="button" value="delete"/>			

To implement modify operation on trainee info refer the following list of screen shots.

Modify Operation

Please enter trainee ID

When admin enters trainee id, in the same page trainee info will be retrieved for modification as shown below in the screen shot

Modify Operation

Please enter trainee ID

Trainee Info

Trainee ID	100
Trainee Name	Rama
Trainee Location	Mumbai
Trainee Domain	JEE
<input type="button" value="update"/>	

After making appropriate entry admin will click update button to reflect changes in Trainee table.

To implement **retrieve** operation on trainee info, refer the following list of screen shots.

Retrieve Operation

Please enter trainee ID	<input type="text"/>	<input type="button" value="retrieve"/>
-------------------------	----------------------	---

When admin enters trainee id, trainee info will be displayed in the same page as shown below in the screen shot.

Retrieve Operation

Please enter trainee ID	<input type="text" value="100"/>	<input type="button" value="retrieve"/>
-------------------------	----------------------------------	---

Trainee Info

Trainee ID	Trainee Name	Trainee Location	Trainee Domain
100	Rama	Mumbai	JEE

Admin selects '*retrieve all*' link to retrieve all trainee info as shown below in the screen shot

Trainees Details

Trainee ID	Trainee Name	Trainee Location	Trainee Domain
100	Rama	Mumbai	JEE
101	John	Mumbai	JEE
102	Aman	Pune	JEE
103	Rehan	Mumbai	.Net
104	Amith	Chennai	Mainframe

Note:

1. Perform appropriate validations on each field for all admin operations including login page
2. If any operation fails admin should be redirected to appropriate error page

Version 2:

Create another Trainee Management application so that all the operations are reflected in the same page as shown in the following screen shots.

Admin logins with valid credentials he will taken to a page as shown below

Trainee Info					
ID	NAME	DOMAIN	LOCATION	OPERATION	
102	Ravan	.Net	Chennai	delete	modify
103	John	.Net	Chennai	delete	modify
106	Cyrus	Mainframe	Pune	delete	modify
104	Hameed	Mainframe	Mumbai	delete	modify
105	John	VnV	Mumbai	delete	modify
100	Rama	JEE	Bangalore	delete	modify
107	Sathia	Common	Bangalore	delete	modify

Add a [New](#) Trainee here

If admin wants to add new Trainee he will click the hyperlink shown at the bottom of the screen shot.

Welcome to Trainee Management System

Enter new trainee details					
Enter Trainee ID	<input type="text" value="0"/>				
Enter Trainee Name	<input type="text"/>				
Enter Trainee Domain	<input type="text"/>				
Enter Trainee Location	<input type="text"/>				
<input type="button" value="add"/>					
Trainee Info					
ID	NAME	DOMAIN	LOCATION	OPERATION	
102	Ravan	.Net	Chennai	delete	modify
103	John	.Net	Chennai	delete	modify
106	Cyrus	Mainframe	Pune	delete	modify
104	Hameed	Mainframe	Mumbai	delete	modify
105	John	VnV	Mumbai	delete	modify
100	Rama	JEE	Bangalore	delete	modify
107	Sathia	Common	Bangalore	delete	modify

Add a [New](#) Trainee here

Admin will click modify hyperlink to make changes to trainee records. The above page appears as shown below in the screen shot.

Trainee Info					
ID	NAME	DOMAIN	LOCATION	OPERATION	
106	Cyrus	Mainframe	Pune	update	modify
102	Ravan	.Net	Chennai	delete	modify
104	Hameed	Mainframe	Mumbai	delete	modify
105	John	VnV	Mumbai	delete	modify
100	Rama	JEE	Bangalore	delete	modify
107	Sathia	Common	Bangalore	delete	modify

Add a [New](#) Trainee here

In the above page admin can enter new details in the text boxes and in the end update button will be clicked to update new details.

If admin selects delete hyperlink that particular record will be deleted.

While creating TMS using above approach take care of the following:

1. Refer class diagrams from previous exercise while designing classes
2. Login screen shot can also be referred from the previous exercise
3. Perform appropriate validations on each field for all admin operations including login page
4. If any operation fails admin should be redirected to appropriate error page

Lab 4. Injecting cross-cutting concerns

Goals	• Using AOP
Time	30 minutes

Using spring and AOP

Refer to Trainee Management System created in the previous lab, implement logging using spring AOP. Logging should take care recording all the admin operations with the timestamp.

Ex: If admin executes inserting new trainee record into database, the logging should log that info in a file called adminoperations.log as shown below:

2013-05-28 09:04:39 INFO Admin executed Add Trainee operation.

Lab 5. Review, Testing and Debugging

Goals	<ul style="list-style-type: none">• Review , Testing and Debugging Spring Applications
Time	90 minutes

Problem Statement 1:

Please do peer to peer review* of the following application and test them by writing spring test cases.

Lab 1: Problem statement-1.2 & 1.3

Lab 3: Problem statement-1 & 2

* Exchange your application with the person sitting next to you and do the review

Problem Statement 2:

You have been provided with some applications by your instructor. Debug those applications and make them working

Appendices

Appendix A: Class Diagrams

Class Diagram 1: Employee.....	4
Class Diagram 2: SBU and Employee.....	5
Class Diagram 3: SBU and Employee (Ver -2)	6
Class Diagram 4: Course Related Classes	8
Class Diagram 5: Trainee related Classes.....	9