Usage Guidelines

Do not forward this document to any non-Infosys mail ID. Forwarding this document to a non-Infosys mail ID may lead to disciplinary action against you, including termination of employment.

Contents of this material cannot be used in any other internal or external document without explicit permission from E&R@infosys.com.

SPRING Basics

Education & Research

Confidential Information

- *This Document is confidential to Infosys Limited. This document contains information and data that Infosys considers confidential and proprietary ("Confidential Information").*
- *Confidential Information includes, but is not limited to, the following:*
  - *Corporate and Infrastructure information about Infosys;*
  - *Infosys' project management and quality processes;*
  - *Project experiences provided included as illustrative case studies.*
  - *< Please list any/all other that is relevant>*
- *Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.*
- *Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.*
- *Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than _(Please insert as applicable)__ without specific written permission of an authorized representative of Infosys.*
- *<Please include this point if applicable> This document also contains third party confidential and proprietary information. Such third party information has been included by Infosys after receiving due written permissions and authorizations from the party/ies. Such third party confidential and proprietary information shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than _(Please insert as applicable)__ without specific written permission of an authorized representative of Infosys.*

Course Objectives

- Spring Introduction
- Spring IOC Concepts
- Introduction to SpEL
- Spring AOP

References

- Spring documentaion

Unit – **1**

Spring Introduction

Spring Introduction

- Spring Framework and its architecture
- Features and uses of Spring Framework
- Spring modules

Introduction

- Spring Framework is an open source Java platform which provides complete infrastructure support in Java application development.
- Spring's usefulness isn't limited to server-side development.
- Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.
- Spring provides elegant integration points with standard and defacto-standard interfaces: Hibernate, JDO, TopLink, EJB, RMI, JNDI, JMS, Web Services, Struts, etc.

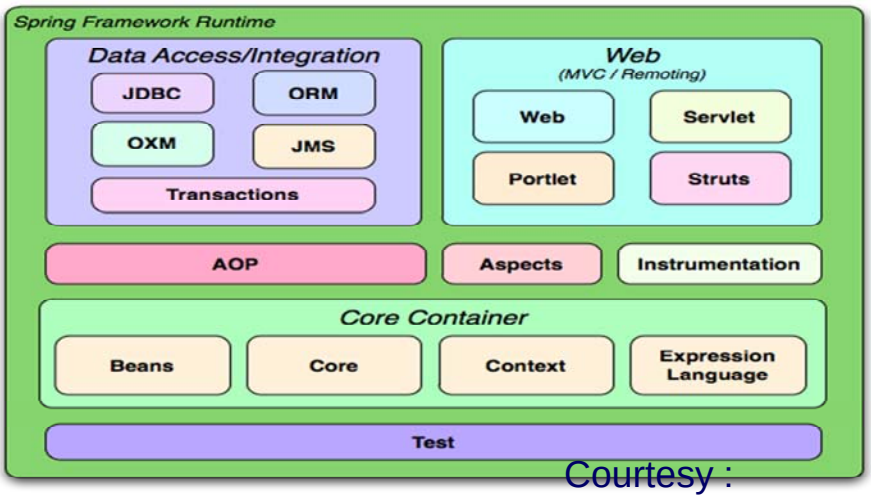
Spring Framework History

- Started 2002/2003 by Rod Johnson and Juergen Holler
- Started as a framework developed around Rod Johnson's book Expert One-on-One J2EE Design and Development
- Spring 1.0 Released March 2004
- 2004/2005 Spring is emerging as a leading full-stack Java/J2EE application framework
- Jolt Productivity Award and a JAX innovation Award.
- The current version of spring is 3.0.x.

Characteristics of Spring

- Lightweight
- Dependency Injection
- Aspect-oriented
- Container
- Framework

Spring modules



Courtesy :
www.springframework.org

Core Container

- It is the most basic part of the framework
- This includes beans, core ,context and SpEL
  - Core & Beans :The IOC & and the DI features are provided by this module. This is the fundamental part of Spring framework.
  - Context :This provides a framework style approch. The Context module inherits from the Beans module.This support for internationalization, event-propagation, resource-loading ,etc. The ApplicationContext interface is the main component of the Context module.
  - SpEL: powerful language for querying and manipulating in the form a object graph at runtime.

Data Access/Integration

- JDBC module :This module provides a JDBC-support that removes set of JDBC code and parsing of database-vendor specific error codes.
- The ORM module: This module provides integration support forJPA, JDO, Hibernate, and iBatis.
- The OXM module :This modules provides an abstraction layer that supports Object/XML mapping implementations specifically for Castor ,JAXB,, XMLBeans, JiBX , XStream
- The JMS module:This module provides the features for producing and consuming messages.
- The Transaction module :This module supports programmatic and declarative transaction management.

Web

- Web Module :*This* provides basic web-oriented integration related features .This also supports Remoting.
- Web-Servlet : This module contains Spring's model-view-controller architecture implementation for web applications.
- Web-Struts : This module contains support classes for integrating a Struts Application within a Spring application
- Web-Portlet Module :This module provides the MVC implementation to be used in a portlet environment

Spring's AOP Package

- It provides an *AOP Alliance*-compliant aspect-oriented programming implementation
- AOP decomposes programs into *aspects* or *concerns*.
- This enables modularization of concerns such as transaction management that would otherwise cut across multiple objects.
- It allowing you to define method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated.
- The *Instrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

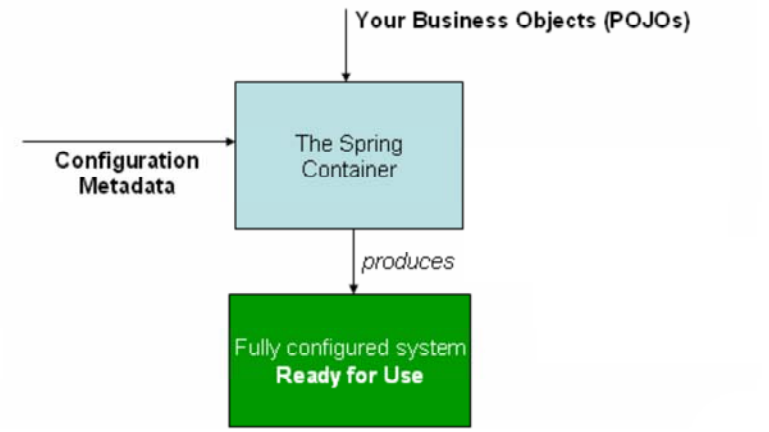
Unit- 2

To introduce Spring IOC Concepts

Core Package

- It is the most fundamental part of the framework
- It provides the IoC and Dependency Injection features.
- The basic concept is the BeanFactory API.

Spring container



•Courtesy :Springframework.org

Spring Container :BeanFactory

- Is the actual *container* which instantiates, configures, and manages a number of beans.
- A BeanFactory is represented by the interface org.springframework.beans.factory.BeanFactory
- The most commonly used simple BeanFactory implementation is
  - **org.springframework.beans.factory.xml.XmlBeanFactory.**
- ApplicationContexts are a subclass of BeanFactory

Bean Factory


```
Resource res = new ClassPathResource("./mypackage/Spring1.xml");
BeanFactory factory = new XmlBeanFactory(res);
```

Spring Context

- Enhances Beans functionality in a more framework-oriented style
- It is a configuration file, provides context information to the Spring frame work
- The ApplicationContext interface builds on top of the BeanFactory and adds other functionality such as easier integration with Spring's AOP features, message event propagation ,resource handling.
- The Spring context includes enterprise services such as JNDI,EJB email, internationalization, validation and scheduling functionality.

Application Context

The most commonly used Application Context are

- ClassPathXmlApplicationContext : This class Loads context definition from an XML file which is located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code .
  - ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

Dependency Injection or Inversion Of Control

- The basic idea of the Inversion of Control pattern or dependency injection is that you no need to create your objects but describe how they should be created.
- The IoC pattern provides better software design that facilitates loose coupling ,reuse, and easy testing of software components

Configuration Metadata

- The Spring IoC container consumes some form of configuration metadata; this informs the Spring container as to how to “create object, configure the object, and assemble [the objects in your application]”.
- This configuration metadata is typically supplied in a simple 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.

An example of the basic structure of XML-based configuration metadata


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

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

Dependency Injection in Spring

- DI exists in two major variants
  - Constructor Injection
  - Setter Injection

Example : Constructor Injection

Consider the following class:

```
package x.y;
public class Company {
public Company(Employee e,Dept d) {
// ...} }
```

Configuration file

```
<beans>
<bean name="c" class="x.y.Company">
<constructor-arg>
<bean class="x.y.Employee"/>
</constructor-arg>
<constructor-arg>
<bean class="x.y.Dept"/>
</constructor-arg>
</bean>
</beans>
```

Setter Injection

- *Setter-based* DI is realized by calling setter methods on your beans after invoking a no-argument constructor to instantiate your bean.

Example

- Find below an example of a class that can only be dependency injected using pure setter injection. **It is plain old Jav**

```
public class GreetingService{
private String greeting;
public void setGreeting(String greeting){
this.greeting=greeting; } }
```

Bean Definition :

```
<beans>
<bean id="greetingService"  class="GreetingService">
<property name="greeting">
        <value>Welcome</value>
</property> </bean> </beans>
```

DEMO 1- Bean Instantiation using Setter DI

NameBean.java

```
package mypackage;

public class NameBean {
String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public void print()
{
System.out.println("The property value is set as:-"+ name);
}
}
```

Spring1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
<bean id="N" class="mypackage.NameBean">
<property name="name">
<value>Abdul Kalam</value></property>
</bean>
</beans>
```

NameClient.java

```
package mypackage;
import java.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
import org.springframework.core.io.*;
public class NameClient {
public static void main(String[] ar)
 {
Resource res = new ClassPathResource("./mypackage/Spring1.xml");
BeanFactory fact= new XmlBeanFactory(res);
NameBean s=(NameBean )fact.getBean("N");
s.print();
} }
```

output

Output is : The property value is set as:-Abdul Kalam

Required Jar files for this application is

- **spring-beans.jar**
- **spring-core.jar**

Spring 3.0 Annotation based Configuration

- In Spring 3 annotation based configuration is possible and the below annotations and class are used for the configuration.
  - AnnotationConfigApplicationContext
  - @Configuration .
  - @Bean.
  - @Component.

Here @Configuration classes are used as input for instantiating an AnnotationConfigApplicationContext. No need to use xml file in this case.

Example : Bean class


```
@Component
public class Employee {
   private int empid;

   public int getEmpid() {
      return empid;
   }
```

Configuration class


```
@Configuration
public class ConfigClass {
  @Bean
  @Scope("prototype")

  Employee employee()
   {
     Employee e=new Employee();
     e.setEmpid(77838);
     return e;
} }
```

Main class


```
public class ClientApp {
   public static void main(String[] args)
 {
AnnotationConfigApplicationContext context =new
AnnotationConfigApplicationContext("ConfigClass.class");

Employee emp = context.getBean(Employee.class);
       System.out.println(emp);
   }
```

Spring Beans Scope

Singleton or not to singleton

- Beans are defined to be deployed in the IOC container in one of two modes:
  - singleton
  - non-singleton.

Singleton

- When a bean is singleton, only one *shared* instance of the bean will be managed.
- All requests for beans with an id matching that bean definition will result in that one specific bean instance being returned.
- In spring container Beans are deployed in singleton mode by default,

Example:

```
<bean id="emp"
class="examples.Employee" singleton= " true "/>
      Or
<bean id="emp"
class="examples.Employee" />
```

Non-singleton or Prototype

- The non-singleton, prototype mode of a bean deployment results in the *creation of a new bean object* every time a request for that specific bean is done.
- This is ideal for situations where for example each user needs an independent user object.

  Example:

  <bean id="emp"

  class="examples.Employee1" singleton="false"/>

Annotation Based Configuration

The scope can be defined in spring 3 by using @Scope("prototype") , @Scope("singleton")

Example :

```
@Configuration
public class ConfigClass {
  @Bean
  @Scope("prototype")

  Employee1 employee()
   {      Employee1  e=new Employee1();
     e.setEmpid(77838);
     return e;   } }
```

Referencing other beans

References to other beans (collaborators)

- The ref element is the last item allowed inside a **<constructor-arg/>** or **<property/>** definition element.
- It is used to set the value of the specified property to be a reference to another bean managed by the container (a collaborator).
  - <ref bean="someBean"/>

Example

```
<bean id="Em" class="mypack.Employee" >
<property
   name="empno"><value>77838</value></property>
<property name="d"><ref bean="de"/></property>
</bean>
<bean id="de" class="mypack.Dept">
<property name="name"><value>ER</value></property>
</bean>
```

AutoWiring Properties

- A BeanFactory is able to *autowire* relations with collaborating beans.
- Beans may be auto-wired (rather than using <ref>)
- The autowiring functionality has five modes
- Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor

Spring's @Autowire

- Spring's @Autowire is used to configure Autowiring in Spring 3.
  - Example

```
public class Employee {
   private int empid;
   private Dept d;
   public Dept getD() {
      return d;    }
@Autowired
   public void setD(Dept d) {       this.d = d;    }

   public int getEmpid() {       return empid;    }

   public void setEmpid(int empid) {       this.empid = empid;    }
}
```

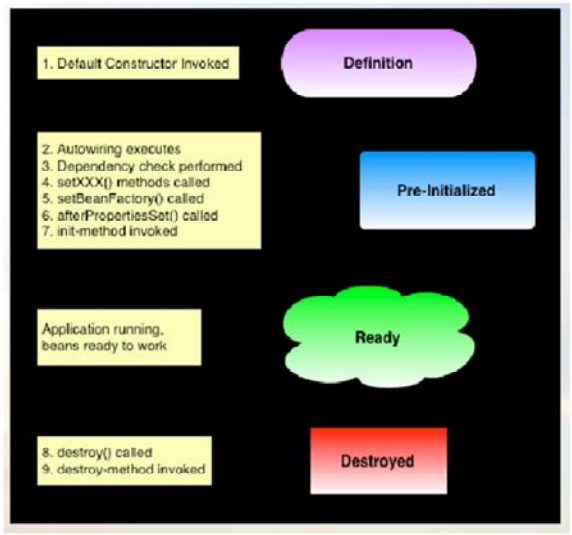
Example


```
public class Employee {
   private int empid;
   private Dept d;

      @Autowired
        Employee(Dept d1)
        {        }
        }
```

The Spring Container Life Cycle

The Bean Factory



Classpath scanning and managed components

Classpath scanning and managed components

- Spring provides a powerful feature of component scanning.
- Classpath Scanning is to detect the candidate components implicitly.
- The classes which compare against a filter criteria and have a corresponding bean definition registered with the container are known as Candidate components.
- The classpath scanning helps to removes the need to use XML to perform bean registration.
- We can use annotations like @Component, AspectJ type expressions, or some custom filter criteria to select which classes will have bean definitions registered with the container.

Classpath Scanning Annotations

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

Example

- Scanning components automatically
  - The basic annotation type that denotes a spring – managed component is @Component.

```
@Component
    public class Emp
    {
    @Autowire
    private Dept d;
    }

    @Service
    Public class Service1
    {}
```

Education and Research

Unit -3

Introduction to SpEL

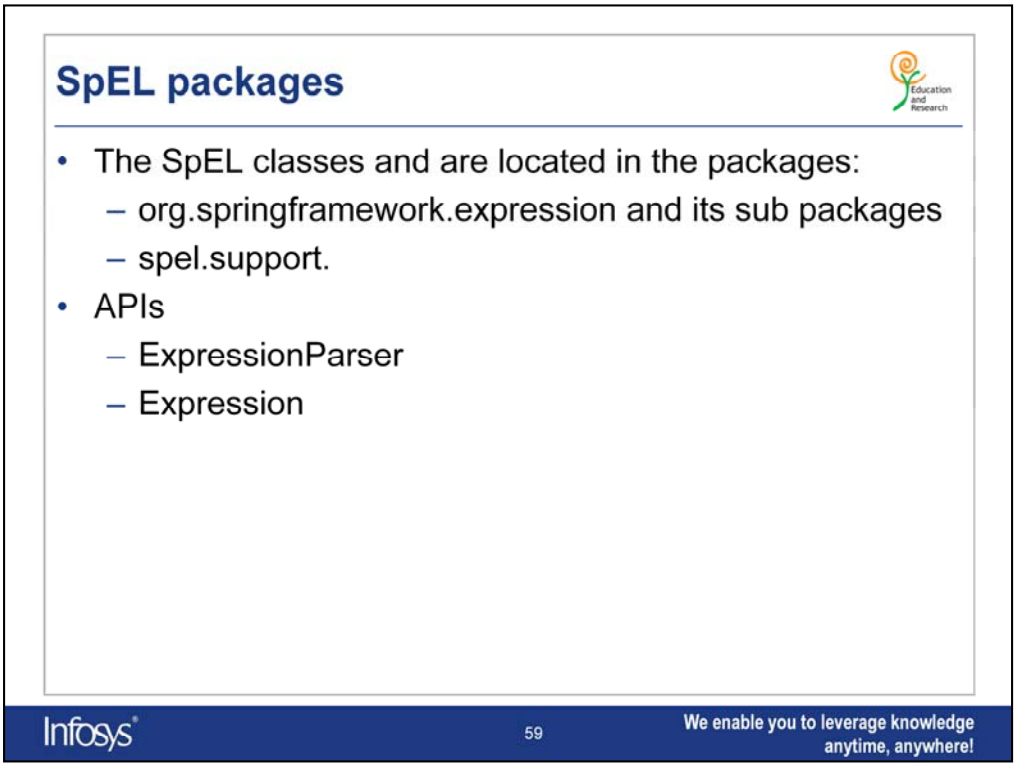
Spring Expression Language

- The **Spring Expression Language** (SpEL for short) is a powerful expression language which supports querying and manipulating an object graph at runtime.
- Most of the language syntax is similar to Unified EL ,also with some additional features.

SpEL

The expression language supports the following functionality
- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, maps
- Method invocation

Relational operators
Literal expressions
- Boolean and relational operators
- Regular expressions
- Assignment
- Calling constructors
- Ternary operator
- Variables
- User defined functions
- Collection projection
- Collection selection
- Templated expressions



The interface ExpressionParser is responsible for parsing an expression string. In this example the

expression string is a string literal denoted by the surrounding single quotes. The interface Expression

is responsible for evaluating the previously defined expression string. There are two exceptions that can

be thrown, ParseException and EvaluationException when calling

'parser.parseExpression' and 'exp.getValue' respectively.

Example

```
ExpressionParser p = new SpelExpressionParser();
Expression e = p.parseExpression("'Hello Spring  World'".concat('!!!')");
String message = (String) exp.getValue();
```

SpEL in config file

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

        <bean id="dept Bean" class="com.infy">
                <property name="id"   value="1" />
                <property name="dname"  value="E&R" />
        </bean>

        <bean id="empBean " class="com.infy">
                <property name="d" value="#{dept}"/>
                <property name="did value="#{dept.id}" />
                <property name="name" value="Rasmi"/>
        </bean>

</beans>
```

Note : Inject "dept" bean into "Employee bean's

Spring EL in Annotation


```
package com.infy;
 import org.springframework.beans.factory.annotation.Value;
 import rg.springframework.stereotype.Component;

 @Component(“empBean”)

public class Customer
 {   @Value("#{dept}")
 private Dept d;;

@Value("#{dept.id}")
 private String did;
 //...   }
@Value(“Rasmi")
 private String name;
 //...   }
```

Spring EL in Annotation


```
import org.springframework.beans.factory.annotation.Value;

 import org.springframework.stereotype.Component;
  @Component("deptBean")

 public class Dept
 {   @Value("1") //inject String directly
private String id;

  @Value("E&R")  //inject interger directly
 private String dname;

   public String getDname()
{ return dname; }
  //... }
```

Education and Research

Unit – 4

Spring AOP

Introduction

- Spring AOP (Aspect Oriented Programming) is meant for applying common behaviors like transaction, logging etc that are required by many classes in an application.
- Spring AOP provides **solution** in a loosely coupled way.

Introduction [Contd…]

- In OOP the key unit of modularity is a class but in AOP the key unit of modularity is an Aspect.

- What is an Aspect?

  Aspects are the cross-cutting concerns that cut across multiple classes. Examples include,

  1. Transaction Management
  2. Logging
  3. Etc.

Terminologies to be familiar with…

- Aspects
- Joint point
- Advice
- Point cut
- Target Object
- Introduction
- Proxy
- Weaving

Terminologies to be familiar with…

- Aspects
  - There are the cross-cutting concerns that cut across multiple classes.
  - Example: Logging
- Joint Point
  - The possible program execution points where the aspects are applied. In spring it is always the method invocation.

Terminologies to be familiar with…

- Advice
  - This represents the action taken by the aspect at a particular joint point.
  - Types
    - Before – Executed before a Joint Point
    - After Returning – Executed after Joint Point returns successfully with out exceptions
    - After Throwing – Executed when the method execution exits by throwing exception
    - After – Similar to finally block in exception handling. This will be executed after the method returns with or without exception
    - Around – This has logics which get executed before method invocation and after the method returns successfully.

Points to Remember

- Before Advice can stop the execution of a Jointpoint only by throwing exception
- After Advice can not change the return value from the joint point.
- AfterThrowing advice will not handle the exception.
- Around advice is very powerful.
  - It can control the method execution
  - It can change the arguments that has to be passed to the method.
  - It can change the return value from the method.
- Aspects can not be the target for other aspects. (i.e) they can not be proxied

Terminologies to be familiar with…

- Pointcut
  - This identifies on what joint points the advice needs to be applied
  - AspectJ Pointcut EL is used for this identification.
- Introduction
  - This helps in adding new interface to the advised object.

Others

- Spring AOP heavily uses Proxy. At runtime the Proxies are created for the business objects with the aspects Linked. This process is called Weaving. The object created is called AOP Proxy or Target Object

Program Perspectives

- Spring AOP can be done in various ways
  - XML based
  - @AspectJ Annotation based (Explained here)

Basic Configuration required:

<aop:aspectj-autoproxy/>

This definition in spring configuration will enable @AspectJ support

Libraries required:

aspectjweaver.jar

aspectjrt.jar

Defining an Aspect

- Any class in the Spring context with @AspectJ annotation is an Aspect

```
@Aspect
public class MyAspect {
}
```

In Spring config file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns: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
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
<aop:aspectj-autoproxy/> <bean id="myaspect" class="mypack.MyAspect"/> </beans>
```

What will be in an Aspect class?

- Like normal Java classes - Methods and fields
- Pointcut declaration
- Advice declaration
- Introduction declaration

Pointcut declaration

- It has two parts
  - Signature with Pointcut Designator [commonly used one is execution]
  - Expression part → Matches a method execution
- Example

```
@Aspect
public class MyAspect {
@Pointcut("execution(* mypack.CalculatePrice.calculate(..))") //expression
public void applyDiscount(){ //Signature
}
}
```

Execution Designator

execution(modifiers-pattern? **ret-type-pattern** declaring-type-pattern?

name-pattern(**param-pattern**) throws-pattern?)

Example: @Pointcut("execution(* mypack.CalculatePrice.calculate(..))")

- Parameters part
  - () represents method with no argument
  - (..) methods with 0 or more arguments
  - (*) method with 1 argument which is of any type
  - (*,int) method with 2 arguments 1st of any type and 2nd of int type
- Modifiers pattern:
  - Example: execution(**public** * *(..))
- declaring-type-pattern – specifies the class part

Other Pointcut Designators

- within(com.infosys.courier.service.*)
- within(com.infosys.courier.service..*)//includes subpackage of service package
- **this**(com.infosys.courier.service.CourierService) //all methods in class of type CourierService
- target(com.infosys.courier.service.CourierService) //all methods of the proxy implemented from CourierService
- args(java.io.Serializable) // any method that takes a single argument that is serializable
- @annotation(org.springframework.transaction.annotation.Transactional) // any jointpoint with @transactional annotation.
- bean(myService) //any jointpoint on bean named myService.

Advice Annotations

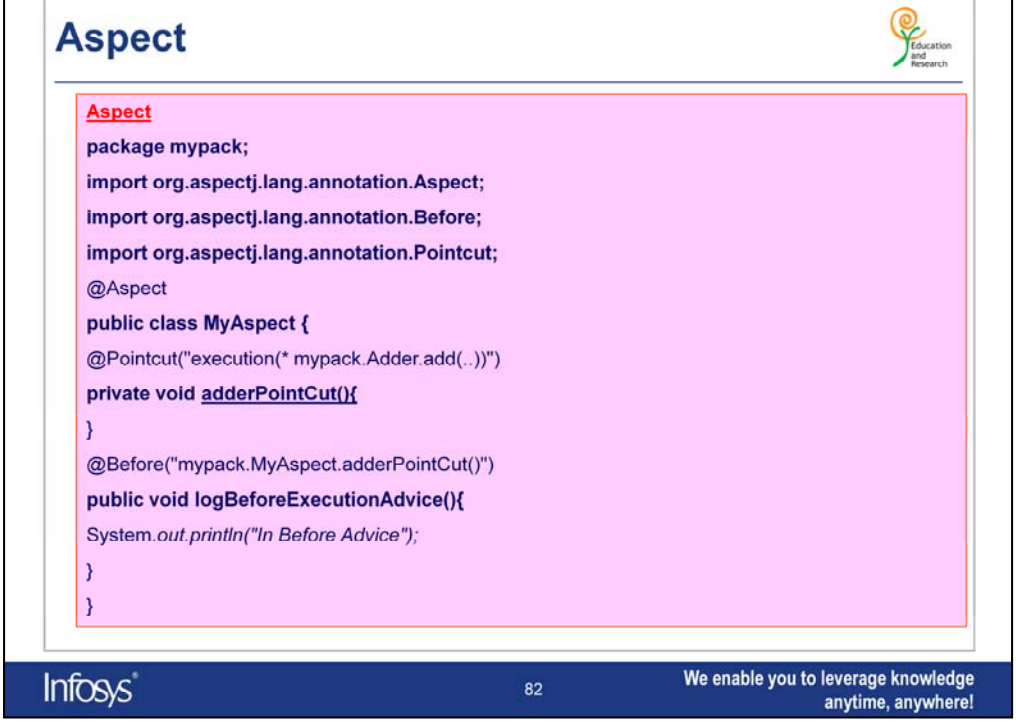
- @Before
- @AfterReturning
- @AfterThrowing
- @After
- @Around

A Simple Example: Adder class to be adviced

```
package mypack;

public class Adder {
public int add(int a,int b){
System.out.println("In add method");
return (a+b);
}
}
```

Aspect



In this example: @ Before advice takes in the Pointcut name "**adderPointCut**". This will identify add method and apply the before advice, **logBeforeExecutionAdvice.**

Config file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns: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
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
<aop:aspectj-autoproxy/>
<bean id="myaspect" class="mypack.MyAspect"/>
<bean id="adder" class="mypack.Adder"/>
</beans>
```

Client Code

```
package mypack;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
public static void main(String[] args) {
    ApplicationContext ctx=new ClassPathXmlApplicationContext("config.xml");
    Adder adder=(Adder)ctx.getBean("adder");
    System.out.println("Result="+adder.add(2, 3));
  }
}
```

Result

```
In Before Advice
In add method
Result=5
```

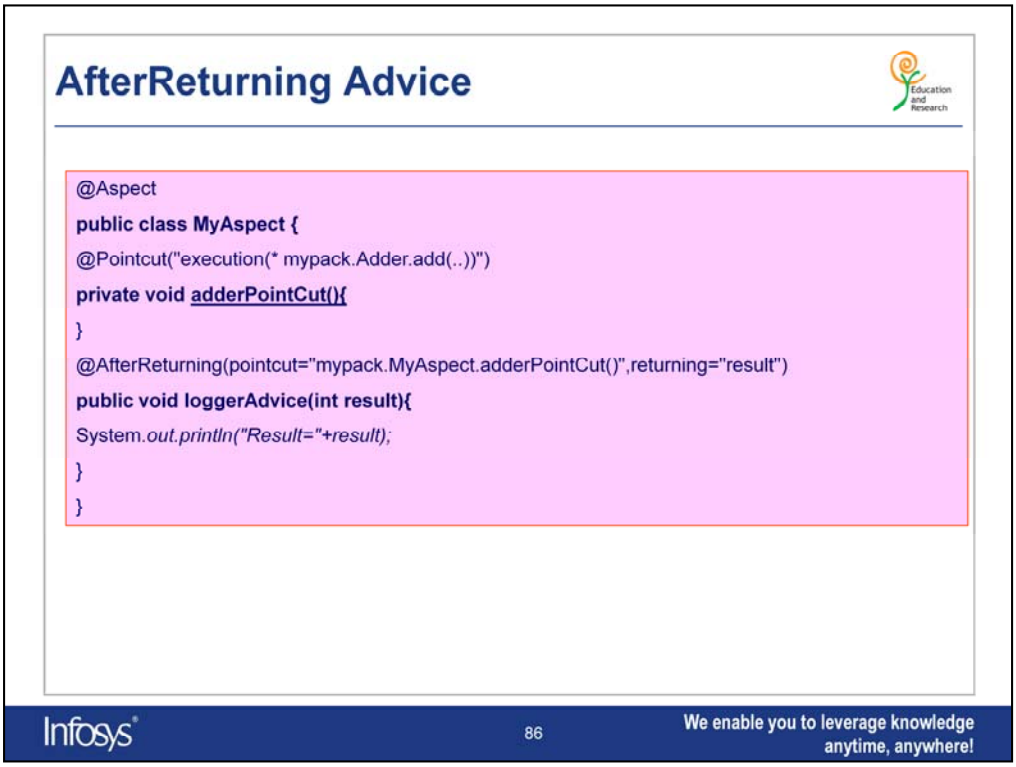
Combining Pointcut declaration and Advice declaration

Aspect

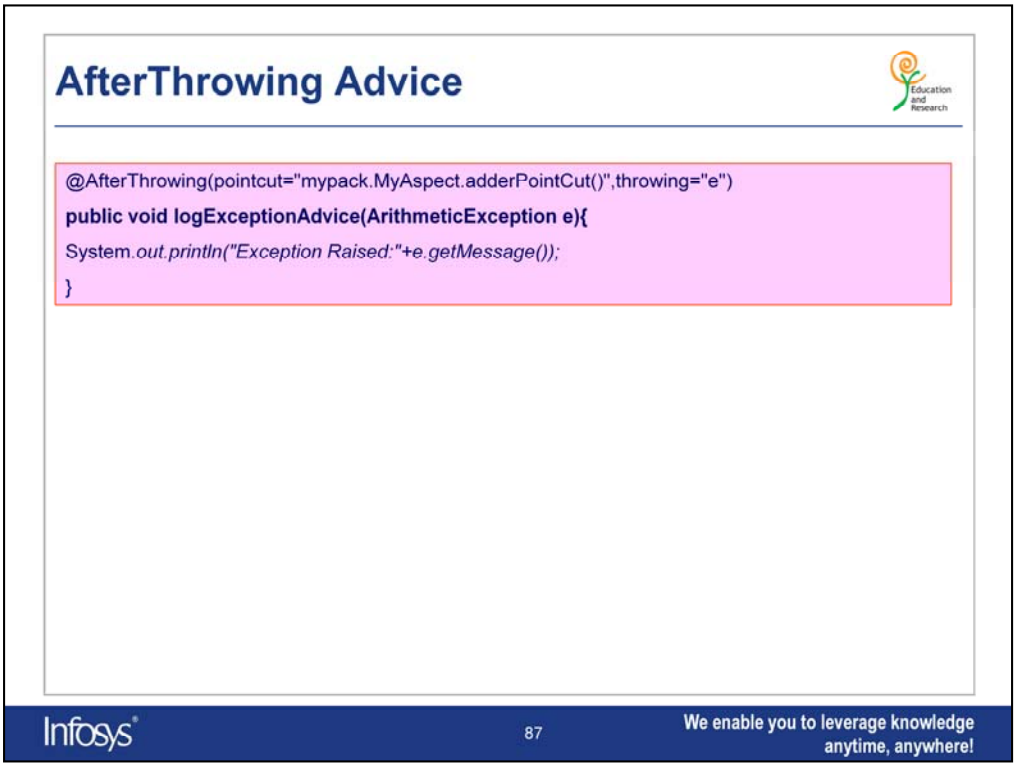
```
package mypack;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class MyAspect {
@Before("execution(* mypack.Adder.add(..))")
public void logBeforeExecutionAdvice(){
System.out.println("In Before Advice_combined approach");
}
}
```



Returning represents the method argument of the advice that holds the return value from the method.



Throwing represents the method argument of the advice that holds the exception raised at the method.

After Advice

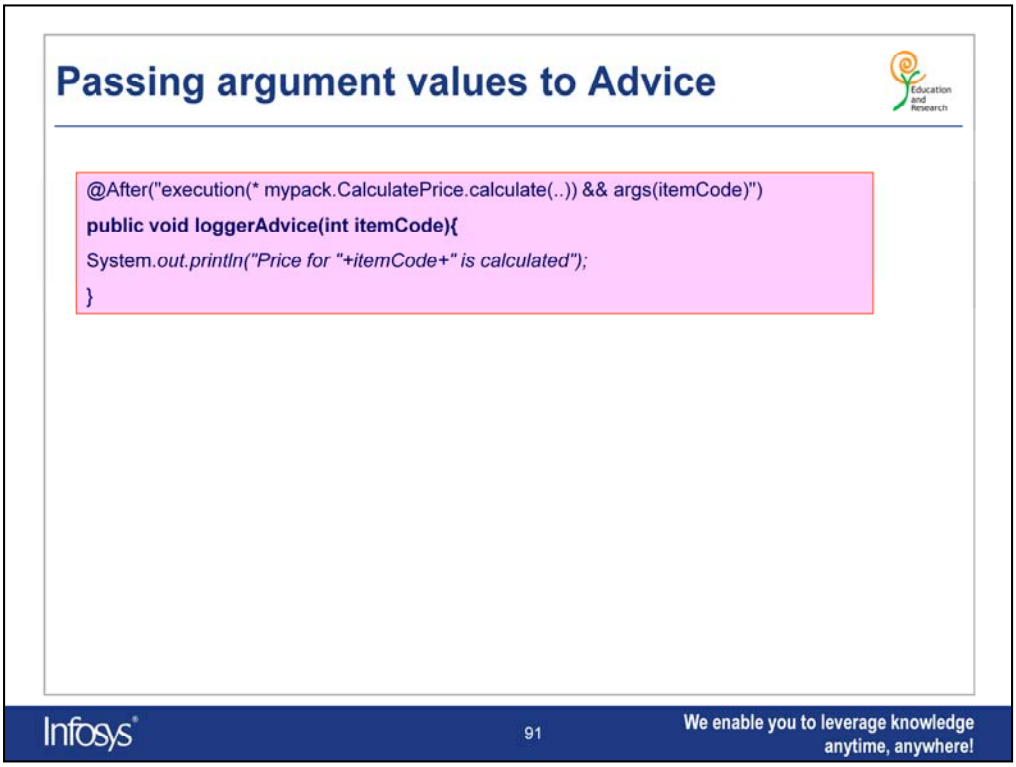

```
@After("mypack.MyAspect.adderPointCut()")
public void finalLoggerAdvice(){
System.out.println("Returning to main");
}
```

Around Advice

```
@Aspect
public class MyAspect {
@Pointcut("execution(* mypack.CalculatePrice.calculate(..))")
public void applyDiscount(){ }
@Around("applyDiscount()")
public Object discountCalculator(ProceedingJoinPoint pjp) throws Throwable{
//insert Pre processing logic if required
Object retVal=pjp.proceed(); // This will invoke the underlying method
//Post processing logic
Object[] args=pjp.getArgs();
double price=((Double)retVal).doubleValue();
int itemCode=((Integer)args[0]).intValue();
if(itemCode==101){  price=(price-(0.3*price)); System.out.println(price); }
if(itemCode==102){  price=(price-(0.1*price)); System.out.println(price); }
return price;
}
}
```

Combining Pointcut Expressions

- Expressions can be combined using
  - &&
  - || and
  - !



From this example:

("execution(* mypack.CalculatePrice.calculate(.)) → identifies calculate method with any number of arguments

args(itemCode)") → adds restriction that the method should have only one integer argument

So calculate(int) is the matching method.

This passes the calculate argument value to the advice argument also.

Introduction

- Consider the scenario where there is a CalculatePriceImpl Class with calculate method which returns the price of the item in Rs. Assume that the class implements CalculatePrice interface.The requirement to add currency conversion methods to the advised object. This can be worked out using Introductions.

CalculatePrice & CalculatePriceImpl

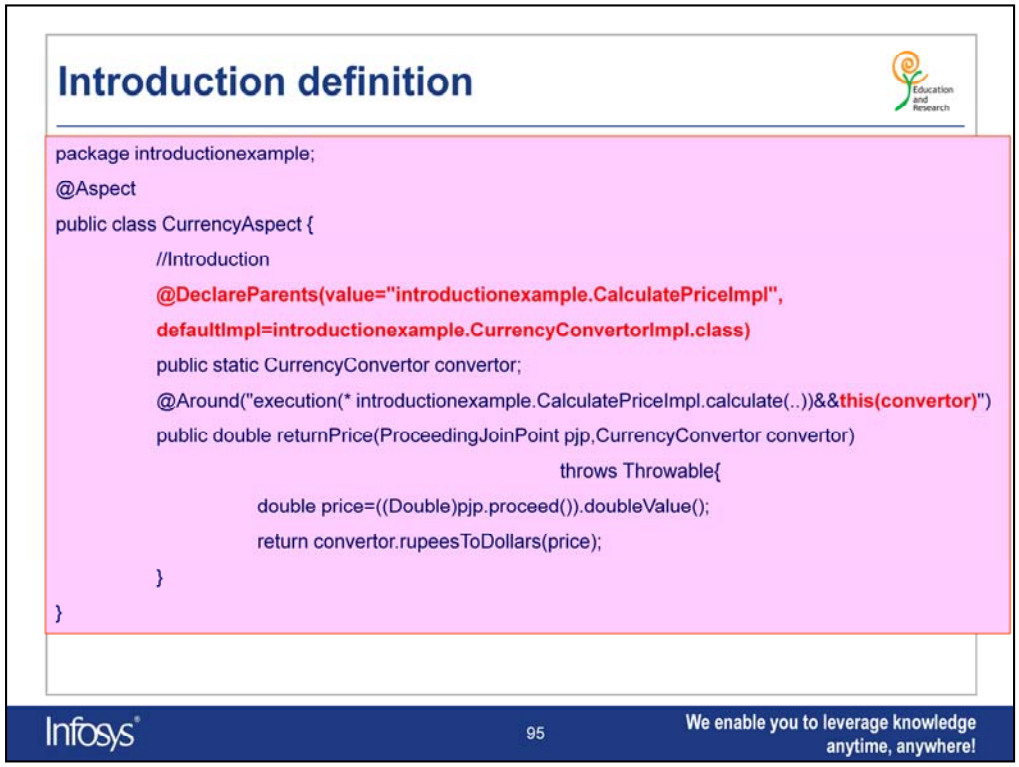
```
package introductionexample;
public interface CalculatePrice {
public double calculate(int itemCode);
}

package introductionexample;
public class CalculatePriceImpl implements CalculatePrice {
public double calculate(int itemCode){
double price=0;
if(itemCode==101)
price=1000.99;
else if(itemCode==102)
price=2000.20;
return price;
}
}
```

CurrencyConvertor and CurrencyConvertorImpl

```
package introductionexample;
public interface CurrencyConvertor {
public double rupeesToDollars(double rupees);
}

package introductionexample;
public class CurrencyConvertorImpl implements CurrencyConvertor {
@Override
public double rupeesToDollars(double rupees) {
return 50*rupees;
}
}
```



@DeclareParents defines the interface to be implemented by the matching bean while generating the proxy. Here bean is CalculatePriceImpl and the method implementation for the interface is available at the implementation CurrencyConvertorImpl. With this configuration, the adviced object will now implement two interfaces

1. CalculatePrice
2. CurrencyConvertor

To the around annotation, this(convertor) condition is added. This matches proxy of type currencyconvertor and maps it to the advice argument.


Education and Research
THANK YOU
Infosys
96
We enable you to leverage knowledge anytime, anywhere!