Basic Spring Framework

Course	Java programming
Trainer	
Designed by	To Chau
Last updated	26-Jul-2011

Course Objectives

- Have knowledge about spring framework
- Understand concepts of aspect-oriented programming.
- Use the Hibernate template to integrate Hibernate and Spring.
- Apply Spring framework in real project.

Contents

- Spring framework introduction
- How to get and setup the spring framework?
- Overview architecture
- Understand IOC/DI
- Understand AOP

Why trainees should learn this topic?

- Spring is a lightweight dependency injection and aspect-oriented container and framework
- From small, simple projects to very complex projects can benefit from this framework
- It is easier to implement an enterprise application.

Introduction

- Spring is an open source framework
- Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.
- Spring is a lightweight dependency injection and aspect-oriented container framework.

How to get and setup the spring framework?

- Download spring libraries from http://www.springsource.org/download
- http://commons.apache.org/downloads/download_loggin
- Create a helloworld project using spring framework(add commons-logging-xxx.jar and spring.jar).

HelloWorld

- Using spring to implement HelloWorld example
- How to inject a bean
- How to get a bean in spring container

Create HelloWorld bean

- Create HelloWorld bean

```
package com.tma.springtraining.helloworld;
public class HelloWorldImpl implements HelloWorld{
    @Override
    public void hello() {
        System.out.println("=========================================");
        System.out.println("Hello world!");
        System.out.println("=========================================");
    }
}
```

Using HelloWorld bean

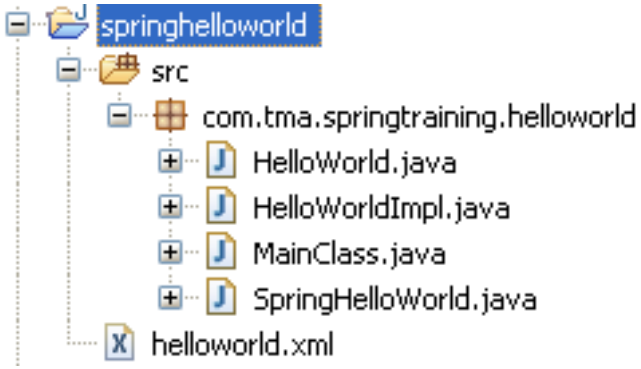
```
package com.tma.springtraining.helloworld;

public class SpringHelloWorld {
    HelloWorld m_helloWorld = null;
    public SpringHelloWorld(HelloWorld helloworld) {
        m_helloWorld = helloworld;
    }
    public void helloSpring() {
        m_helloWorld.hello();
        System.out.println("Welcome to spring fwk!");
    }
}
```

Define the beans in configuration file

```
<?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="helloworld" class="com.tma.springtraining.helloworld.HelloWorldImpl"/>
    <bean id="springHelloworld" class="com.tma.springtraining.helloworld.SpringHelloWorld" autowire="constructor"/>
</beans>
```

Run HelloWorld example

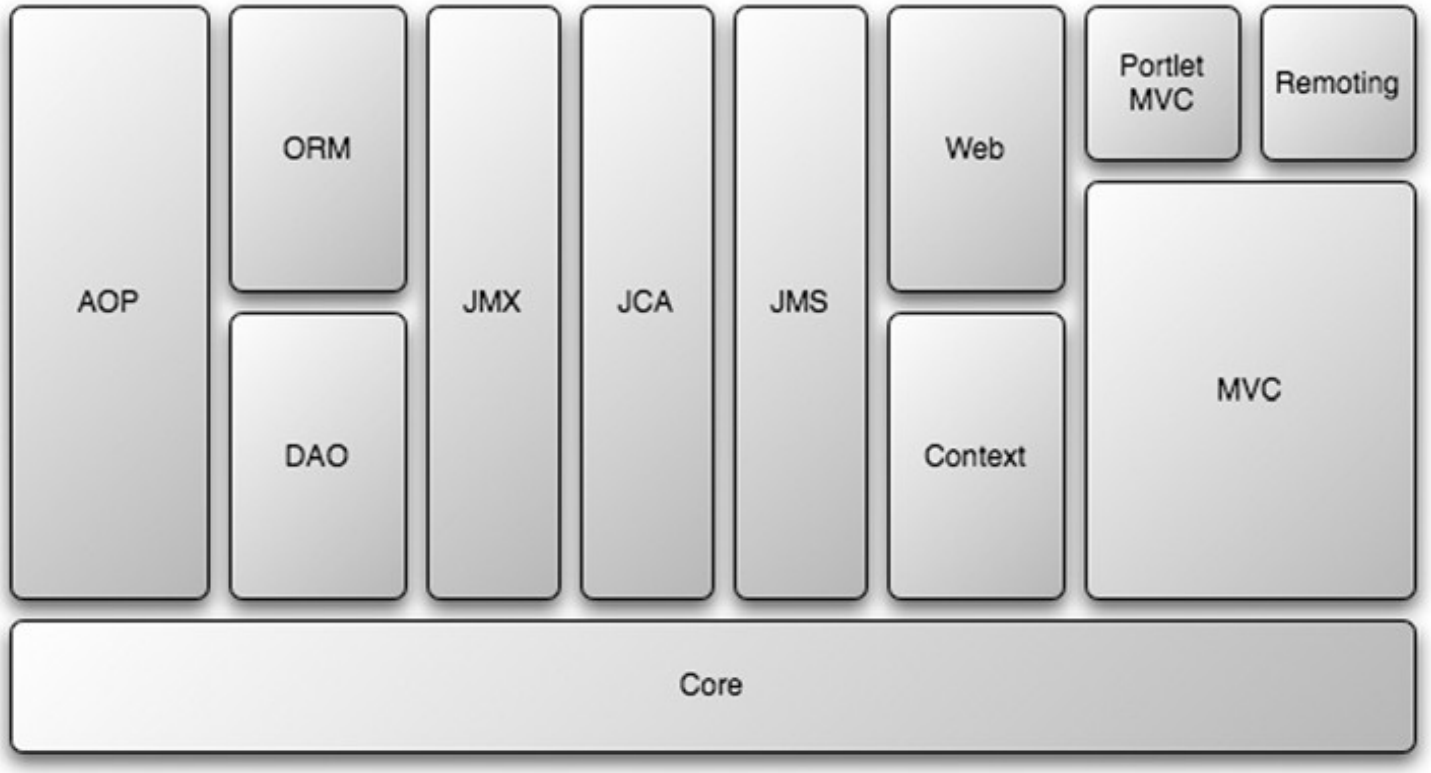
```
package com.tma.springtraining.helloworld;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
public class MainClass {
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("helloworld.xml"));
        //Very basic hello world
        HelloWorld hw = (HelloWorld)factory.getBean("helloworld");
        hw.hello();
        //Using spring IOC/DI for spring hello world
        SpringHelloWorld shw = (SpringHelloWorld)factory.getBean("springHelloworld");
        shw.helloSpring();
    }
}
```



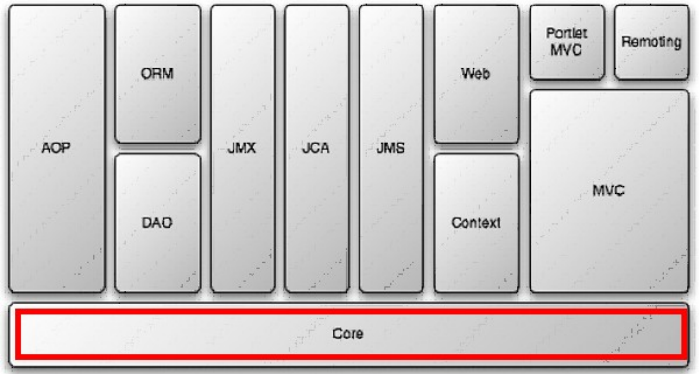
Architecture Overview

- Understand the role of the main components in spring

Architecture Overview

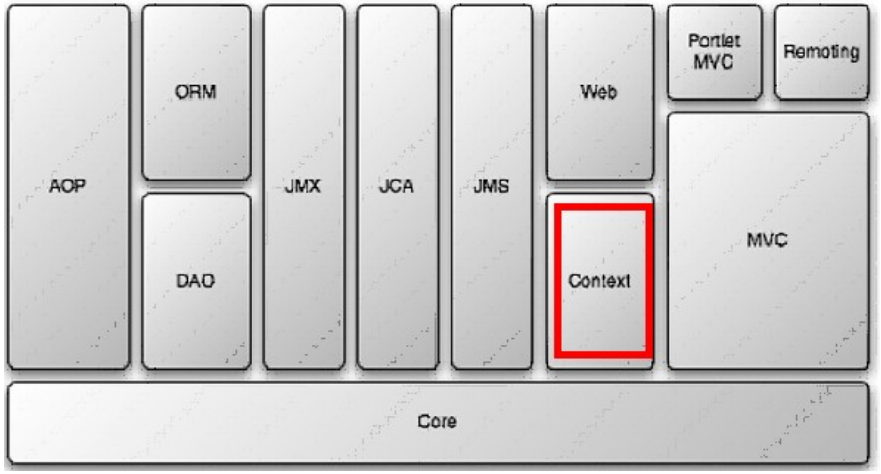


The core container



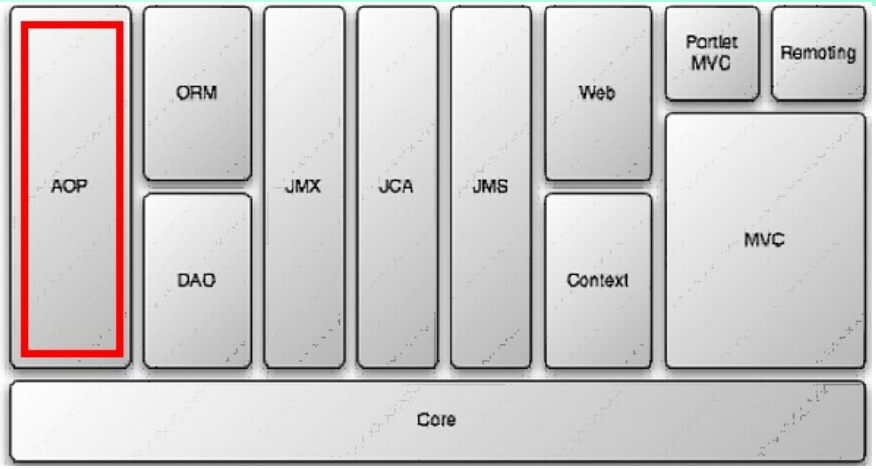
- The container defines how beans are created, configured, and managed.
- Core module is the core container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave.

Application context module



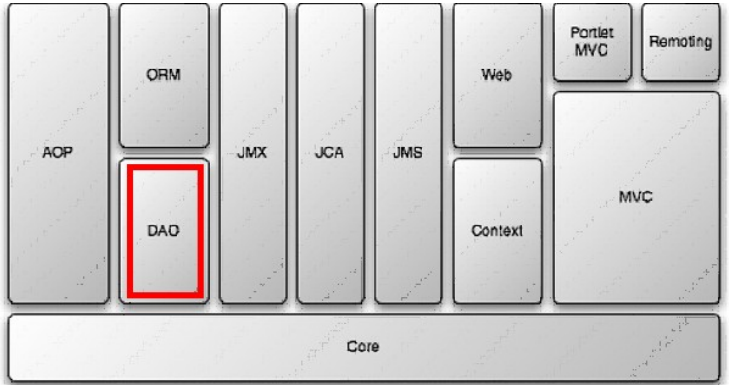
- The core module's BeanFactory makes Spring a container, but the context module is what makes it a framework.
- Support for internationalization (I18N) messages, application lifecycle events, and validation. In addition, this module supplies many enterprise services such as email, JNDI access, EJB integration, remoting, and scheduling.

AOP



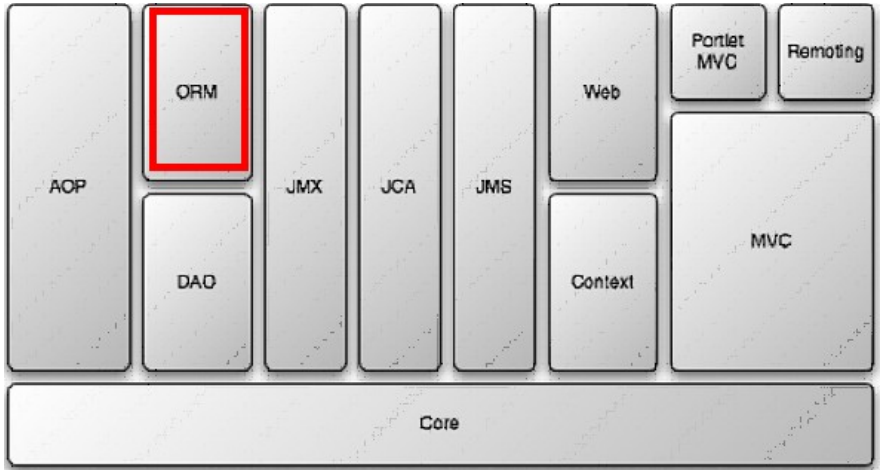
- ***Spring's AOP module***
  - Spring provides rich support for aspect-oriented programming in its AOP module.
  - This module serves as the basis for developing your own aspects for your Spring enabled application. Like DI, AOP supports loose coupling of application objects.
  - With AOP, however, applicationwide concerns (such as transactions and security) are decoupled from the objects to which they are applied.

JDBC abstraction and the DAO module



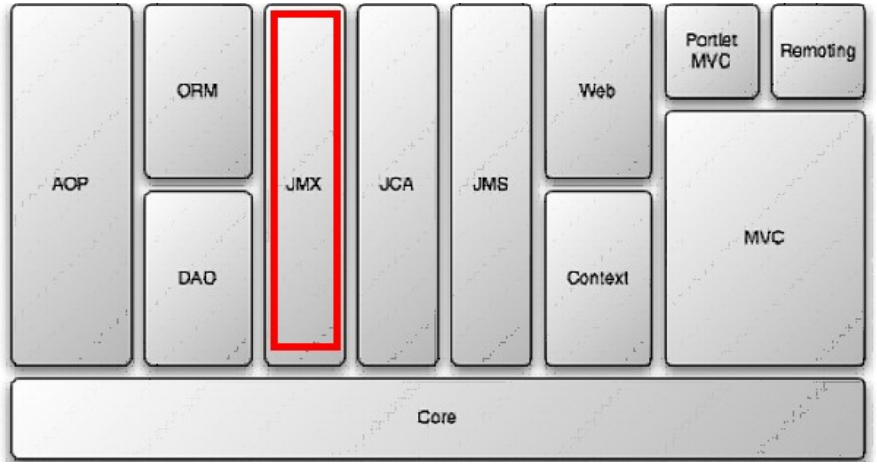
- Spring's JDBC and Data Access Objects (DAO) module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources.
- In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application.

ORM(Object-relational mapping)



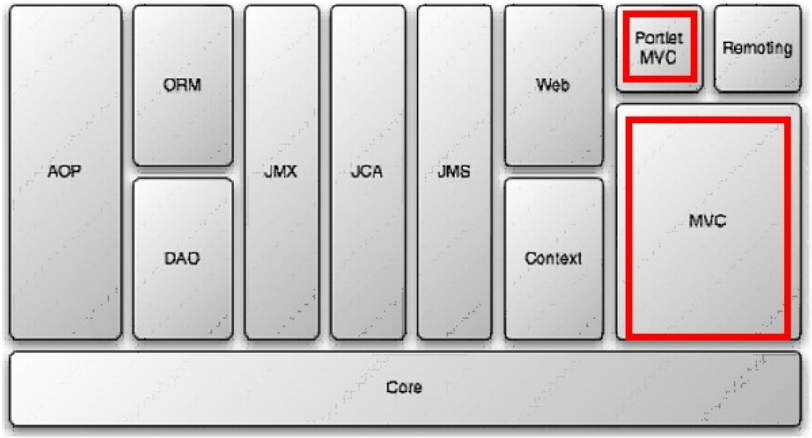
- Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions.
- Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps.
- Spring's transaction management supports each of these ORM frameworks as well as JDBC.

JMX



- ***Java Management Extensions (JMX)***
  - Spring's JMX module makes it easy to expose your application's beans as JMX MBeans.

The Spring MVC framework

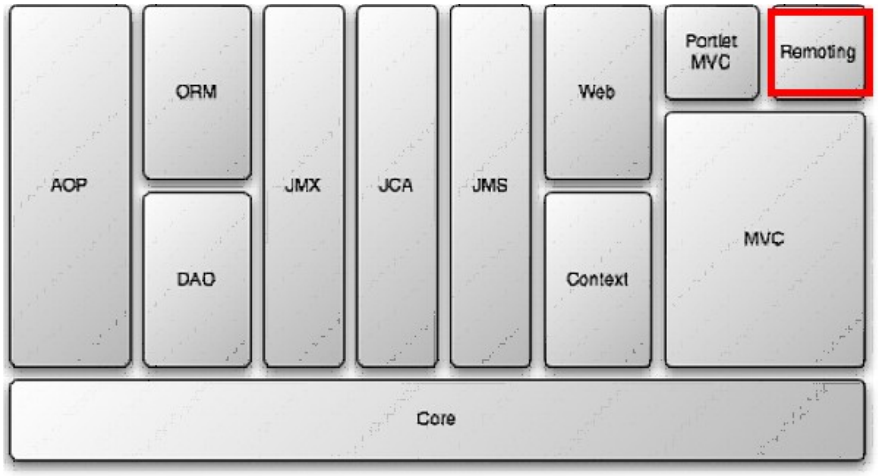


- Supports Apache Struts, JSF, WebWork, and Tapestry etc.
- It also has its own MVC frameworks

Spring Portlet MVC

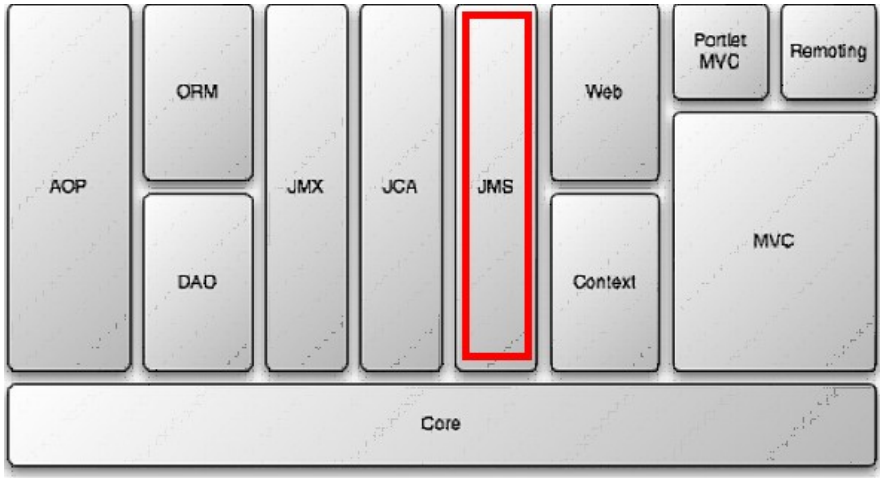
- Spring Portlet MVC builds on Spring MVC to provide a set of controllers that support Java's portlet API.
- Spring MVC and Spring Portlet MVC require special consideration when loading the Spring application context. Therefore, Spring's web module provides special support classes for Spring MVC and Spring Portlet MVC.
- It also contains integration support with Apache Struts and Java-Server Faces (JSF).

Remoting



- Spring's remoting support enables you to expose the functionality of your Java objects as remote objects.
- Or if you need to access objects remotely, the remoting module also makes simple work of wiring remote objects into your application as if they were local POJOs.

JMS

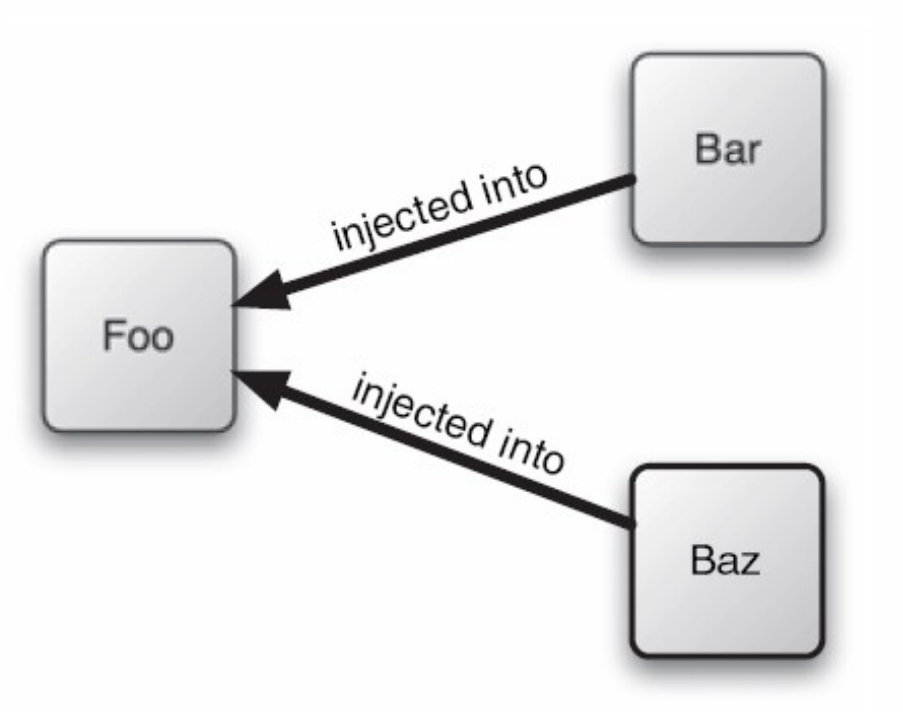


- ***Java Message Service (JMS)***
  - Spring's Java Message Service (JMS) module helps you send messages to JMS message queues and topics.

IOC/DI

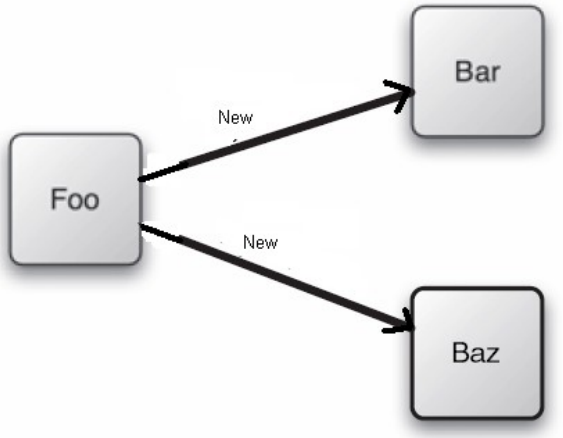
- Understand what IOC/ID is in spring

IOC/DI



Understanding IOC/DI

- Each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies).
- This can lead to highly coupled and hard-to-test code.



First Example

- First Example(Test reference objects indirectly)

```
public class HolyGrailQuest {
  public HolyGrailQuest() {}

  public HolyGrail embark() throws GrailNotFoundException {
    HolyGrail grail = null;
    // Look for grail
    …
    return grail;
  }
}
```

```
public class KnightOfTheRoundTable {
  private String name;
  private HolyGrailQuest quest;

  public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();
  }

  public HolyGrail embarkOnQuest()
      throws GrailNotFoundException {
    return quest.embark();
  }
}
```

Unit test for KnighOfTheRoundTable

- Unit test for KnighOfTheRoundTable class

```
import junit.framework.TestCase;
public class KnightOfTheRoundTableTest extends TestCase {
  public void testEmbarkOnQuest() throws GrailNotFoundException {
    KnightOfTheRoundTable knight =
        new KnightOfTheRoundTable("Bedivere");
    HolyGrail grail = knight.embarkOnQuest();
    assertNotNull(grail);

    assertTrue(grail.isHoly());
  }
}
```

- Unit test for HolyGrailQuest class -> before even get started, you realize that Your KnightOfTheRoundTableTest case indirectly tests HolyGrailQuest

Second Example

- Second Example(test ability)

```
public class ChatSession {
    ServerConnection m_connection = null;
    public ChatSession() {
        m_connection = new ServerConnection("192.168.98.123");
    }
    public void send(String message){
        m_connection.send(message);
    }
    public String receiveMessage(){
        // Wait for message from server and return
        String message = m_connection.getMessage();
        // Process, check or validate message before return
        // ..........
        message = preProcess(message);
        return message;
    }
    private String preProcess(String message) {
        // Does some check locally
        return null;
    }
}
```

```
public class ServerConnection {

    public ServerConnection(String ip) throws NetworkConnectionException{
        // Open a Socket connection to the server
        // ...
    }
    public void send(String message) {
        // Send message to server
    }

    public String getMessage() {
        return null;
    }
}
```

Unit test for ChatSession

- Implement unit test
  - Now we want to implement unit test for our ChatSession class.

    When running unit test the server is not available, the build machine is not connected to network -> can not implement unit test.

Decoupling with interfaces

- Hide implementation detail behind interfaces -> how to retrieve the interface?

```
public class ServerConnection implements Connection{

    public ServerConnection(String ip) throws NetworkConnectionException{
        // Open a Socket connection to the server
        // ...
    }
    public void send(String message) {
        // Send message to server
    }

    public String getMessage() {
        return null;
    }
}
public class ChatSession {
    Connection m_connection = null;
    public ChatSession() {
        try {
            m_connection = new ServerConnection("192.168.98.123");
        } catch (NetworkConnectionException e) {
        }
    }
```

→ this is not better than before.

Giving and Taking

- Giving and Taking

```
public class ChatSession {
    private Connection m_connection = null;
    public ChatSession() {
    }
    public void send(String message){
        getConnection().send(message);
    }
    public String receiveMessage(){
        // Wait for message from server and return
        String message = getConnection().getMessage();
        // Process, check or validate message before return
        // ..........
        message = preProcess(message);
        return message;
    }
    private String preProcess(String message) {
        // Does some check locally
        return null;
    }
    public void setConnection(Connection connection) {
        m_connection = connection;
    }
    public Connection getConnection() {
        return m_connection;
    }
}
```

Implement unit test

```
import junit.framework.TestCase;
public class ChatSessionTest extends TestCase{
    public void testReceiveMessage(){
        ChatSession session = new ChatSession();
        session.setConnection(new DummyConnection());
        session.receiveMessage();
    }
    class DummyConnection implements Connection{
        @Override
        public String getMessage() {
            return "test message";
        }
        @Override
        public void send(String message) {

        }
    }
public class ChatSessionTest extends AbstractDependencyInjectionSpringContextTests {
    private Connection m_connection; ——— will be setted by spring container at runtime
    @Override
    protected String[] getConfigLocations() {
        return new String[]{"testApplicationContext.xml"};
    }
    public void testReceiveMessage(){
        ChatSession session = new ChatSession();
        session.setConnection(m_connection);
        session.receiveMessage();
    }
}
```

→ this is all IOC/DI about

Spring Container

```
public static void main(String[] args) {
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource("hello.xml"));
    IocExample ex = (IocExample) factory.getBean("iocExample");
    ex.callDeviceServiceMethod();

    m_beanFactory = new FileSystemXmlApplicationContext("hello.xml");
    Performer performer = (Performer) m_beanFactory.getBean("performer_steven");
    try {
        performer.perform();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- BeanFactory(XmlBeanFactory)
  - Knows about many objects within an application
  - Create associations between collaborating objects as they are instantiated.
  - Takes part in the lifecycle of a bean
  - Etc…
- ApplicationContext(FileSystemXmlApplicationContex)
  - More advantages -> is preferred to use.

Summary of IOC/DI

- Dependences are given at creation time
- Loose coupling
- Easy to test

Q&A

AOP

- Understand AOP in spring

AOP

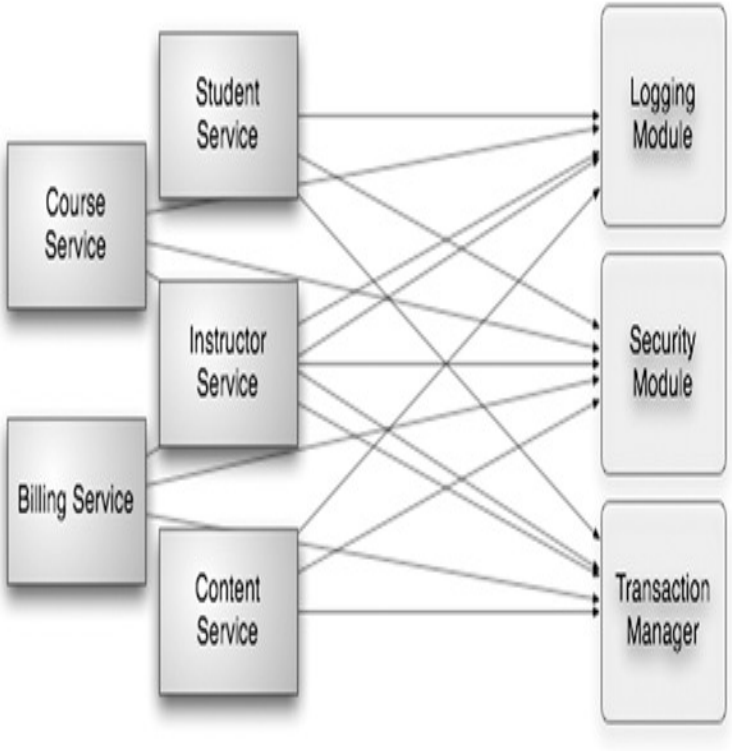
Aspect oriented programming enables you to capture functionality that is used throughout your application in reusable components.

Aspect-oriented programming is often defined as a programming technique that promotes separation of concerns within a software system.

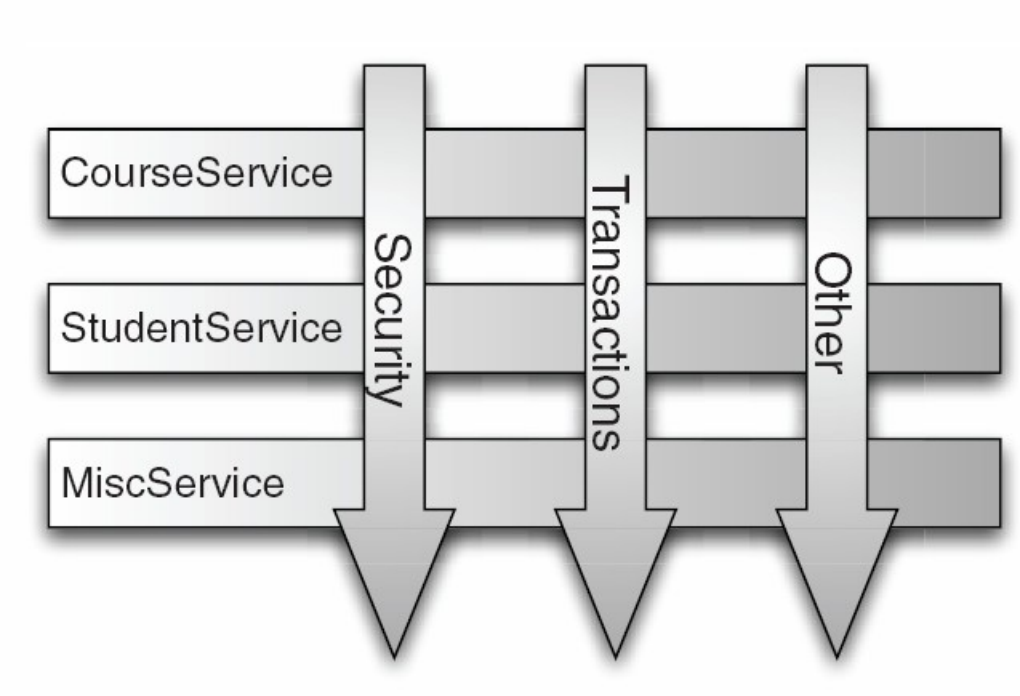
Systems are composed of several components, each responsible for a specific piece of functionality.

Understand AOP

```
Public void commonFunction(){
  If(alreadyLogin()){
       beginTransaction();
       try{
        insertFirstThingIntoDb();
        insertDetailOfFirstThingIntoDb();
       }catch(Exception ex){
        rollback();
       }
       endTransaction();
  }
}
```

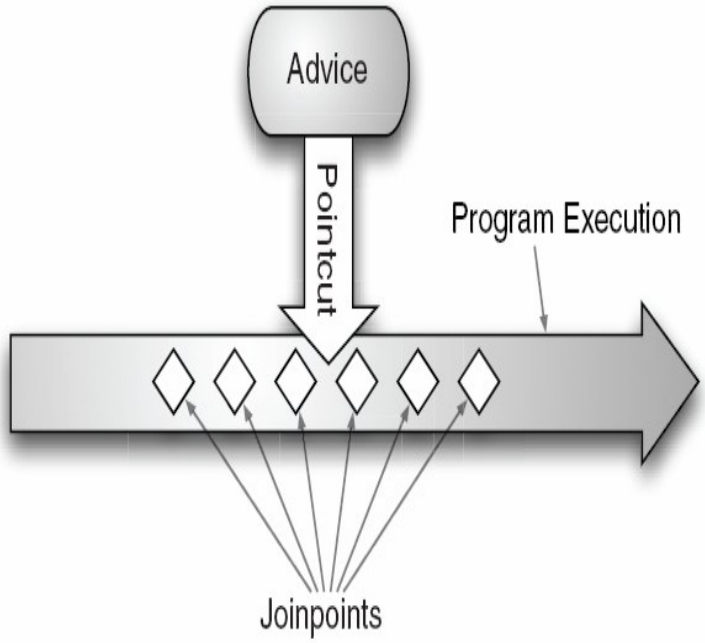


AOP


CourseService
StudentService
MiscService
Security
Transactions
Other

AOP Jargon

- **Advice**: defines both the *what* and the *when* of an aspect (applied before, after or when an exception occurs).MethodBeforeAdvice, AfterReturningAdvice,ThrowsAdvice
- **Pointcut**: defines where, one pointcut definition matches one or more joinpoints. ex: execution(* *.perform(..))
- **Joinpoints**: A *joinpoint* is a point in the execution of the application where an aspect can be plugged in (method, field, constructor etc..). Spring only supports method joinpoints



Advices

- Advice:
  - Before advice: MethodBeforeAdvice
  - After returning advice:AfterReturningAdvice
  - After Throwing advice:ThrowsAdvice
  - Around advice:is effectively before, after-returning, and after-throwing advice. In spring it is defined in MethodInterceptor.

Example of advice

```
package com.springinaction.springidol;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;

public class AudienceAdvice implements
    MethodBeforeAdvice,
    AfterReturningAdvice,
    ThrowsAdvice {

  public AudienceAdvice() {}

  public void before(Method method, Object[] args, Object target)
      throws Throwable {
    audience.takeSeats();
    audience.turnOffCellPhones();
  }

  public void afterReturning(Object returnValue, Method method,
      Object[] args, Object target) throws Throwable {
    audience.applaud();
  }

  public void afterThrowing(Throwable throwable) {
    audience.demandRefund();
  }

  private Audience audience;
  public void setAudience(Audience audience) {
    this.audience = audience;
  }
}
```

Implements three types of advice

Invokes before method

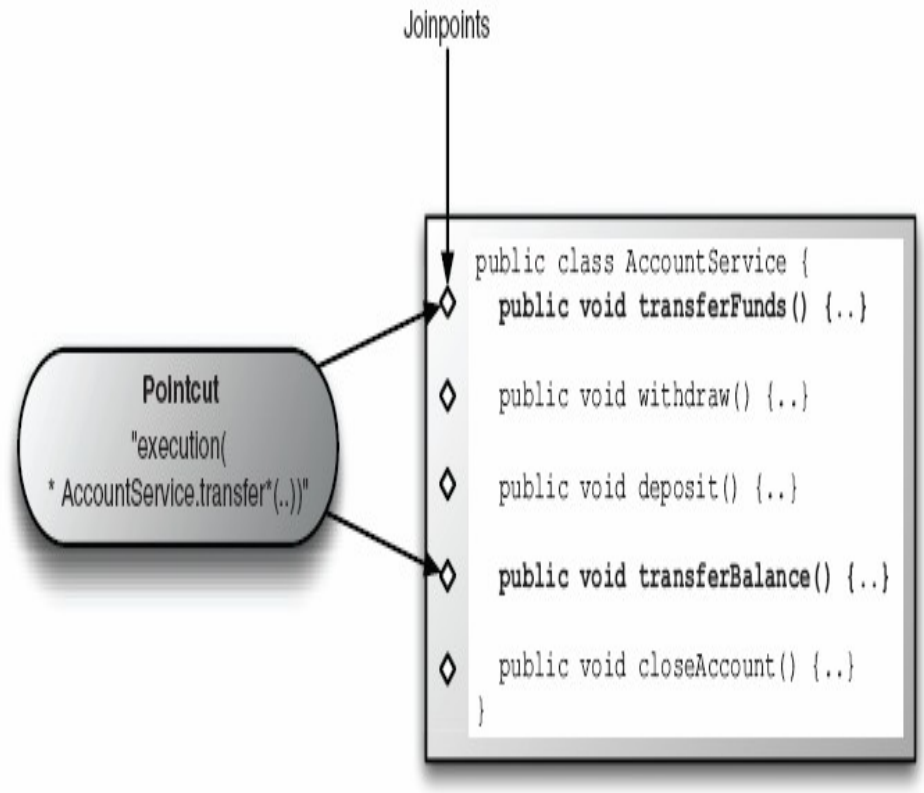
Executes after successful return

Executes after exception thrown

```
<bean id="audienceAdvice"
    class="com.springinaction.springidol.AudienceAdvice">
  <property name="audience" ref="audience" />
</bean>
```

Pointcuts and Advisor

- Defining pointcuts and advisors
- Two of the most useful pointcuts are regular expression pointcuts and AspectJ expression pointcuts.



Declaring a regular expression pointcut

- Define a pointcut using JdkRegexpMethodPointcut

```
<bean id="performancePointcut"
    class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="pattern" value=".*perform" />
</bean>
```

- Associate advice and pointcuts

```
<bean id="audienceAdvisor"
    class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="pointcut" ref="performancePointcut" />
</bean>
```

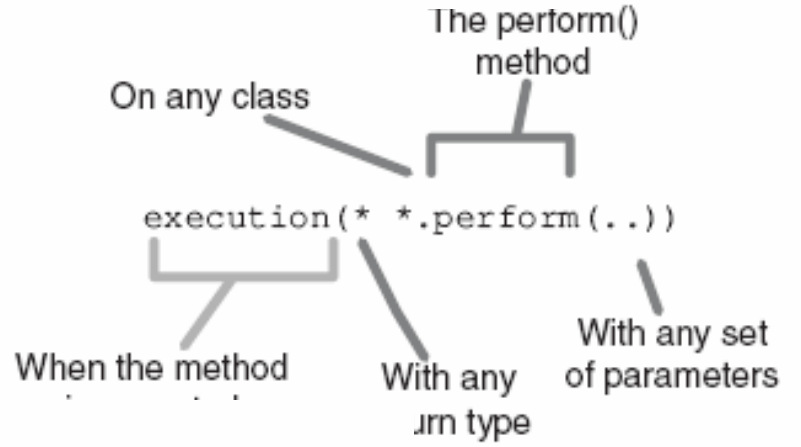
- Compiling pointcut with an advisor

```
<bean id="audienceAdvisor"
    class="org.springframework.aop.support.
        ➥ RegexpMethodPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="pattern" value=".*perform" />
</bean>
```

Defining AspectJ pointcuts

- Define pointcut

```
<bean id="performancePointcut"
    class="org.springframework.aop.aspectj.
            ➥ AspectJExpressionPointcut">
  <property name="expression" value="execution(* Performer+.perform(..))" />
</bean>
```

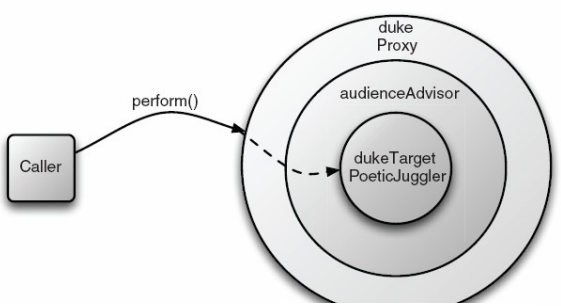


- Associate pointcut and advice

```
<bean id="audienceAdvisor"
    class="org.springframework.aop.aspectj.
            ➥ AspectJExpressionPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="expression" value="execution(* *.perform(..))" />
</bean>
```

Using ProxyFactoryBean

- Advisors completely define an aspect by associating advice with a pointcut.
- But aspects in Spring are proxied.
- You'll still need to proxy your target beans for the advisors to take effect.




```
<!-- Using ProxyFactoryBean proxy-->
<bean id="duke" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="dukeTarget" />
    <property name="interceptorNames" value="advisor" />
    <property name="proxyInterfaces" value="com.tma.learnspring.example.aop.Performer" />
</bean>
<bean id="dukeTarget" class="com.tma.learnspring.example.aop.Juggler"></bean>
```

Notes and best practices using this technique

- Logging
- Security component
- Translations
- Common pre-checking

Summary

- AOP
  - 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.

Q&A

References

- http://www.springsource.org/documentation
- Manning.Spring.in.Action.2nd.Edition.Aug.2007.pdf

Document Revision History

Date	Version	Description	Revised by
26 Jul 2011	1.0	First version	To Chau